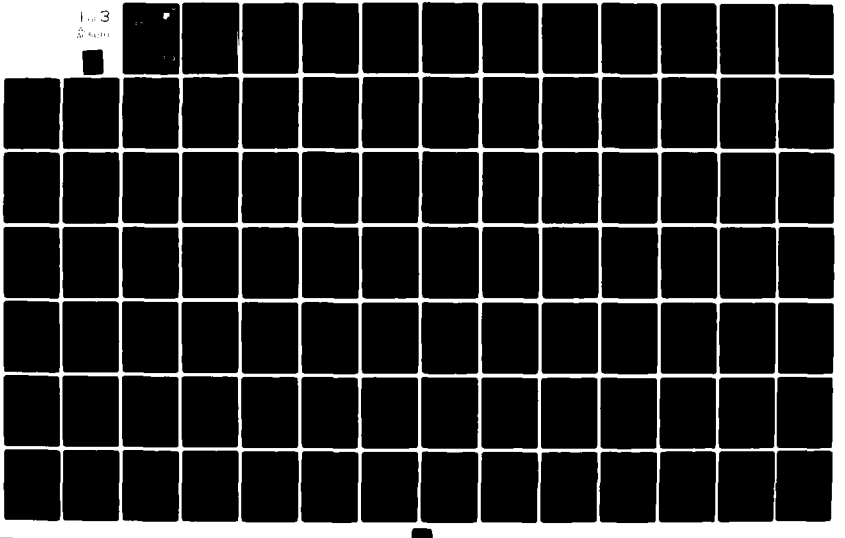
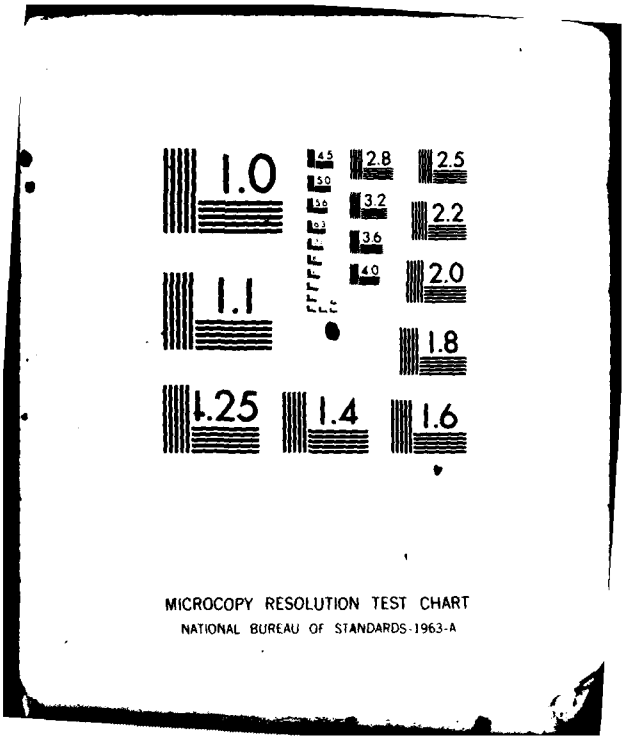
AD-A115 810 HONEYWELL INC BLOOMINGTON MN CORPORATE COMPUTER SCIE--ETC F/G 9/2
CONCURRENT SYSTEM DESCRIPTION LANGUAGE.(U)
FEB 82 W T WOOD, H K BERG, S H YU F30602-80-C-0295
UNCLASSIFIED RADC-TR-82-3 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-82-3
Final Technical Report
February 1982

CONCURRENT SYSTEM DESCRIPTION LANGUAGE

Honeywell Inc.

William T. Wood
Helmut K. Berg
Steve H. Yu
William R. Paulsen

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-82-3 has been reviewed and is approved for publication.

APPROVED: Armand A. Vito

ARMAND A. VITO
Project Engineer

APPROVED: J. J. Marciniak

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-3	2. GOVT ACCESSION NO. AD-A115010	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) CONCURRENT SYSTEM DESCRIPTION LANGUAGE		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 11 Sep 80 - 11 Sep 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) William T. Wood William R. Paulsen Helmut K. Berg Stone H. Yu		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0295
9. PERFORMING ORGANIZATION NAME AND ADDRESS Honeywell Inc. Corporate Computer Sciences Center 10701 Lyndale Ave. S., Bloomington MN 55420		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 25290108
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEA) Griffiss AFB NY 13441		12. REPORT DATE February 1982 13. NUMBER OF PAGES 240
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Armand A. Vito (COEA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Concurrent Processing System	Computational Model	Communication
Design	Performance	Data Types
Specification	Abstraction	Program
Description	Refinement	Machine
Architecture	Decomposition	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes an effort to develop a concurrent system description language (CSDL). The development of the notation is based on studies of approaches to modeling, designing, decomposing, describing, analyzing, and deriving simulations of concurrent processing systems. The devised notation is based on a computational model which represents systems as a collection of communicating concurrent activities. CSDL takes an architectural approach viewing concurrency as a relationship among activities,

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

rather than a concurrent programming approach viewing concurrency as a control construct. The model does not distinguish between hardware and software realizations. The notation integrates statements about the performance of concurrent processing systems. The basic principles of CSDL include data abstraction and refinement, hierarchies of abstract machines and their decomposition, representation of system requirements and design solutions for all system components.

System definitions in CSDL include both the specification of requirements on the system and the description of a system design satisfying these requirements. Definitions in CSDL are given at the system level of observation; they aid the designer in developing and evaluating concurrent systems at several levels of abstraction, without being biased by realization considerations of such systems. When used as a modeling tool, CSDL accepts definitions of functional, behavioral, and structural requirements as well as performance constraints, and supports the specification of system properties, the derivation of design descriptions, the stepwise refinement and decomposition of system designs, and the verification and analysis of system designs. In this report, the use of CSDL is demonstrated for the design of a simple concurrent system including a complete communication subsystem, the definition of an operating system kernel, and the performance specification for an interactive multi-programmed computer system. The CSDL syntax is formally defined in the form of a BNF description and syntax diagrams.

ACKNOWLEDGEMENT

The authors wish to express their gratitude to W. Franta for his contriutions to the development of performance analysis concepts and an experimental framework for performance analysis of CSDL system definitions, and to A. Pizzarello for his contributions to the development of structuring and specification concepts and techniques. The authors also thank T. Remple for his help with the initial formalization of the CSDL snytax and S. Bieglari for his help in editing the formal syntax definition. R. Kain's review of this report was helpful in enhancing its clarity. The guidance and valuable suggestions provided by the contract monitors A. Vito and Lt. D. Lopez are greatly appreciated.

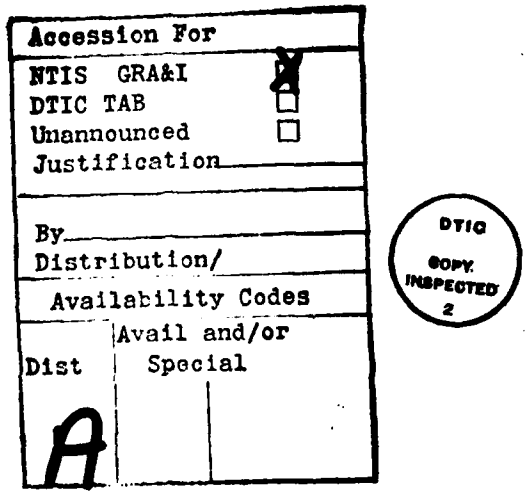


TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

SECTION 1
INTRODUCTION

1.1 INTRODUCTION TO THE PROBLEM

In recent years a number of advances in the electronics industry have had an impact on the design and development of information processing systems. In particular, the rapidly decreasing cost of hardware has accelerated the trend toward development of systems consisting of collections of components cooperating to accomplish a single task. Such systems are referred to as concurrent processing systems. Concurrent systems promise modularity, expandibility, and reliability, which are some of the most desirable properties of information processing systems. However, these advantages of concurrent processing have not been clearly demonstrated. For example, distributed processing systems are a particular type of concurrent processing system. While most hardware problems in building distributed processing systems have been overcome, overhead of 50% and more when compared to centralized systems has been observed in their application [MARI80]. Among the problems leading to these deficiencies are:

- o The geographical dispersion of system components
- o The distribution of functions and data within the system
- o The interactions of concurrently operating components

The state of the art in concurrent system technology has not kept up with the advances in the electronics industry. It can be argued that a thorough understanding of concurrent system concepts is lacking. This lack causes difficulties in designing concurrent systems, including their modeling, description, and analysis.

Concurrent system design is in its infancy, and technologies are needed which supplant ad hoc development techniques and integrate precisely defined concepts and theories into systematic engineering disciplines [BERG81]. To this end, concepts need to be discovered which clarify the understanding of concurrent systems. Once such concepts are identified, they need to be incorporated into methodologies which guide the design of efficient and well structured concurrent processing systems.

Concepts, theories, and languages, together with scientific principles of design, do not make up a methodology by themselves. These theoretical foundations are hard to understand, and in addition, there is a need for working procedures which can be followed in the different phases of the design and development process. Furthermore, the successful application of working procedures needs to be supported by appropriate tools, thus evolving the methodology into an engineering discipline.

An integral part of a design methodology is a notational tool for producing system documentation which is complete, clear and unambiguous. Such a notation constitutes the basis for a system design methodology. There is a clear need for a concurrent system description language as a first step towards a concurrent system engineering discipline. The development of such a language involves the identification and definition of conceptual foundations for concurrent system design as a basis for a collection of techniques guiding the design of concurrent systems.

1.2 CHARACTERISTICS OF CONCURRENT SYSTEMS

The problems associated with the design and description of concurrent processing systems relate to the fundamental characteristics of such systems. In the following subsections, the general characteristics of concurrent systems are discussed in terms of concurrent system architectures and properties of concurrent systems.

1.2.1 Overview

Concurrent processing systems consist of multiple activities operating asynchronously on global information. Concurrent processing concepts have evolved along the distinct lines of multiprogramming on the one hand and computer networking on the other. The first category is associated with centralized systems, the second with decentralized systems. Developments in the area of multiprogramming were essentially motivated by performance considerations, whereas the idea of computer networking was centered around the need for sharing geographically distributed resources.

One of the most important issues in the design of concurrent systems is the communication among activities used to effect the cooperation required to realize the

overall system function. This issue arises at all levels in concurrent systems; an example is the seven layer communication model for Open System Interconnection Architecture defined by the International Standards Organization [ISO80]. Processor multiplexing and the use of shared storage provide the means for communication among activities in centralized systems. Depending on the characteristics of communication among activities, decentralized systems can be grouped into two broad categories -- loosely coupled and tightly coupled systems. Loosely coupled systems are characterized by low bandwidth communication (compared to the rate at which activities change their internal state), whereas tightly coupled systems exhibit high bandwidth communication. A corollary to low bandwidth communication in loosely coupled systems is the lack of quickly accessible shared storage (address space). Consequently, in such systems the interaction between activities is realized by message exchange. The development of concurrent systems of this kind has been based on developments in computer networking, and has led to the evolution of distributed processing. Figure 1 is an overview of concurrent processing systems.

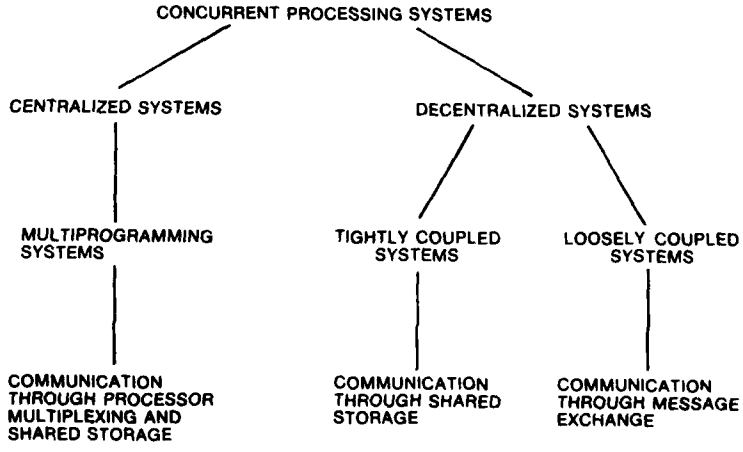


Figure 1. Concurrent Processing Systems

1.2.2 Concurrent System Architecture

The definition of a concurrent system involves identification of the concurrent activities, distribution and sharing of the information which they access, mechanisms for communication among them, and their computational structure.

Concurrent Activities -- Concurrent activities involve sequences of actions which can mutually overlap in time. An activity may itself be composed of a set of concurrent actions, thus leading to a hierarchy of concurrent activities.

Organization of Information -- The organization of information refers to how information is made available to activities. Options concerning information organization include:

- Shared global data which can be accessed directly by concurrent activities, as in the case of tightly coupled systems
- Local data which can only be shared through message-based communication among concurrent activities, as in the case of loosely coupled systems

Communication Mechanisms -- Communication in concurrent systems can be represented by a logical communication network. The topology of such a communication network can be defined by statically or dynamically established connections between activities. Information flow among activities can be specified by:

- Activity names [HOAR78], i.e. direct addressing of the sender or receiver
- Port names [SILB81], [COOK80], where a port is owned by an activity
- Names of message buffers [AMBL77].

Three fundamental synchronization modes have been identified for concurrent systems:

- Synchronous - handshaking between the sender and the receiver [HOAR78]
- Asynchronous - sender is not blocked [FELD79]
- Remote Invocation - sender is blocked until it receives some reply message from the receiver [HONE80]

Computational Structure -- The interactions, precedence relationships, and flow of information among concurrent activities constitute the computational structure of concurrent systems. The computational structure defines the dynamic behavior of the system. Dynamic creation or annihilation of activities is a part of the interaction among activities within the computational structure.

1.2.3 System Properties

System designs must not only meet the functional requirements but also possess performance properties. Therefore, realistic system specifications must include some information about performance requirements and how they are distributed among the system components. The development of tightly coupled systems has in the past been primarily motivated by high performance requirements for certain compute bound problems. Loosely coupled systems provide the potential for high performance when the inherent parallelism of a problem is mapped onto an appropriate organization of concurrent activities. The definition of an appropriate communication network for a loosely coupled system is crucial to attaining the goal of high performance.

Reliability and fault tolerance (or graceful degradation) are achieved by detecting error conditions and invoking certain recovery mechanisms. The recovery procedures either reconfigure the system and isolate the failed components or roll back the system to some previous state called a regeneration point. The realization of these properties is more complex in concurrent systems than in sequential systems because of the difficulties involved in defining regeneration points for a set of concurrent activities. Further complications may arise in loosely coupled systems where the global state may be distributed over the nodes of the network and thus is not readily available to the individual nodes.

Confidence in the correctness of a design can be established either by the application of formal proof techniques or by the use of informal and intuitive arguments. Certifying correctness is more complex for concurrent systems than for sequential systems because the global state can be altered simultaneously by many activities. In loosely coupled systems arguments about correctness are generally based on analysis of the message sequences. Certification of correctness for concurrent system has two major aspects:

- o Functional - input-output relationship and job determinacy, i.e., the output of the system depends only on the state of the input data and is independent of the communication delays in the system.

- o Operational - liveness and freedom from deadlocks, i.e., the system does not enter a state from which no activity can proceed.

Security and protection are of natural concern in concurrent systems, as in any other practical system. Some of the protection problems unique to concurrent systems are:

- o Protection of information during communication among activities

- o Protection of global information from unprivileged concurrent activities

- o Protection of information in the local domain of each activity from all other activities

1.3 DEFINITION AND SCOPE OF CSDL

The Concurrent System Description Language (CSDL) presented in this report is a notation for describing concurrent processing systems. It is based on a computational model which represents systems as a collection of communicating concurrent activities. The underlying model does not distinguish between hardware and software realizations of activities. The notation integrates statements about the performance of concurrent processing systems. The development of the notation is based on studies of approaches to modeling, designing, decomposing, describing, analyzing, and deriving simulations of concurrent processing systems.

System descriptions in CSDL represent concurrent systems at the system level of observation. At the same time they aid the designer in developing and evaluating concurrent systems at several levels of abstraction without being biased by aspects of realizations of such systems. For example, it is possible

- To lay out the topology of a concurrent system and to investigate different methods of communication among system components without being biased by realization characteristics of particular communication protocols.

- To specify system performance with respect to particular functional requirements or system topologies without being bound by realization decisions concerning hardware, firmware and software components.

The objectives of CSDL imply that its development is primarily concerned with the following areas:

- Concepts for modeling concurrent systems

- Languages for defining concurrent systems

- Techniques for analyzing concurrent systems

When used as a modeling tool, CSDL accepts definitions of functional, behavioral, and structural requirements as well as performance constraints to be met by a concurrent system. These requirements need to be re-expressed in terms of the model of concurrent systems underlying CSDL. This re-expression of requirements definitions is necessary in order to apply techniques for:

- The specification of properties

- The derivation of design descriptions

- The verification and analysis of system designs

- The stepwise refinement and decomposition of system designs

Thus, the interest in requirements definition concentrates on the comprehension of the types of requirements pertinent to concurrent systems, rather than on methods for deriving and validating requirements definitions.

Several concurrent system implementation languages have been proposed in the literature, such as ADA [HONE80], Modula [WIRT77], Concurrent Pascal [BRIN73], CSP [HOAR78], etc. These languages are based on specific implementation models for such systems. In particular, language constructs have been conceived which accommodate different forms of communication schemes such as message passing or direct remote invocation. Representative implementation language constructs differ in terms of:

- The effect on the flow of control they impose on sending and receiving nodes
- The nature of synchronization assumed for the invocation of such constructs
- The structure they impose on programs which use them

It is not clear what implementation language construct forms are the most viable. However, this is of minor importance to the development of CSDL because it should express what is to be accomplished, in a fashion which allows a realization to use whatever constructs are deemed appropriate, in a reasonably unbiased fashion. Thus, the interest in implementation languages is in the compatibility of the model of communication through information exchange used in CSDL, with implementation models for message passing or direct remote invocation, rather than in different organizations of node software as imposed by these languages.

1.4 REPORT ORGANIZATION

The body of this report is arranged in six sections. This first section has identified the problem, introduced basic notions of concurrent systems, and defined the scope of the Concurrent System Description Language contract. Section 2, System Description Elements, addresses the problems relating to modeling, designing and analyzing concurrent processing systems. In Section 3, CSDL Conceptual Foundations, the general approach taken in this effort is presented. Section 4, CSDL Definition, defines the computational model used to exhibit system behavior, the notation for describing concurrent processing systems in terms of this model, and a set of organizational concepts which involve notions of system structure and documentation format.

Examples of the application of CSDL to concurrent system design are demonstrated in Section 5, An Illustration of System Definition in CSDL. An extension of the computational model for performance, associated language constructs, and examples of performance specifications in CSDL are presented in Section 6, System Performance Specification in CSDL.

Four appendices are included. Appendix A summarizes the CSDL notation and document organization in tabular form. Appendices B and C formally define the CSDL syntax in Backus-Naur form and syntax diagrams, respectively. Appendix D contains a CSDL definition of an operating system kernel.

SECTION 2
SYSTEM DESCRIPTION ELEMENTS

In this section, problems relating to modeling, designing and analyzing concurrent processing systems are identified. These problems need to be addressed in the selection and evaluation of conceptual foundations of CSDL and techniques for using CSDL. The organization provides a framework for discussing issues of concurrent system description language research in subsequent sections.

The thrust of the problem statement presented here is that the basic problems associated with the design and description of concurrent processing systems center around:

- The concepts which are needed to model all aspects of such systems
- The integration of these concepts into different models
- Notations for expressing the concepts of the different models
- Strategies for analyzing system designs in terms of these concepts and notations

2.1 MODELS

In this section the notion of a model for information processing systems is introduced and defined, and an argument for the necessity of models in any engineering discipline is presented. Some issues which must be addressed when developing models for information processing systems are discussed under the two categories of usage in the development process and modeling scope.

2.1.1 The Notion and Use of Models

To understand and think about complex systems, people need to develop highly organized frameworks of concepts concerning various aspects of the systems which are

of interest. The "aspects of interest" are rather pragmatically defined to be what one is currently trying to do to or with the system. If the activity is design, for example, then one needs to understand the required function, how the system interacts with its environment, how parts interact to accomplish the function, and how efficiently it uses resources. If the activity is certification and validation, then one must understand how to infer functional, operational, and performance information from a description of the structure and component parts of the system as well as how to extract the information required for usable simulations and emulations. Since a design is usually expressed in more or less abstract terms, while a realization is made up of real components, one needs to be able to understand the idealizations behind the design and how their realization affects the results of analysis and validation.

In the realm of technology, scientists develop frameworks for understanding general problem areas while engineers use their experience and creativity to design systems that solve particular problems. The engineer must also understand the relationship between the "theoretical" assumptions of the scientists and the "practical" realities of available materials and techniques in order to create feasible designs. It follows, then, that such frameworks must be developed before the specification, design, analysis, and realization of concurrent systems can become part of an engineering discipline. Such a framework will be referred to as a model.

A model is defined to be a conceptual structure consisting of:

- A set of primitive terms and relations among the terms
- A set of rules for defining or deriving new terms and relations from existing ones
- A set of axioms concerning the primitive terms and relations

Given these components of a model, statements involving both primitive and derived terms and relations can be deduced from the axioms by using the accepted rules of logic. For a model to be usable, there must also exist rules which define the association between terms in the model and objects in the real world and between relations in the model and relationships among objects in the real world. In the established sciences the relations and terms have been chosen both for their relevance

to the phenomena being modeled and for their convenient mathematical expression; this has allowed well-developed mathematics to be used for communication as well as analysis.

2.1.2. Issues an Information Processing Model Must Address

In the previous paragraphs it was argued that some model of information processing must be developed to be able to specify and describe any particular concurrent system. Here the scope of such a model is presented. The issues group naturally into those associated with the inherent characteristics of information processing systems and those associated with the use of models in the design process.

Design Process -- The process by which people design systems is a dynamic, iterative one. Requirements are analyzed, tentative solutions are created, and refinements are introduced. Models are used both to guide the problem-solving activity and to facilitate rigorous analysis. Analysis of a refined solution may show the solution is inadequate and that either further refinements or a totally different solution is required.

In established engineering disciplines one finds that, in fact, several models are routinely used during the design process. For example, in electrical engineering one model encompasses the structure of connections of a circuit and the essential behavior of its discrete components. This model is based on the physics of time-varying electromagnetic fields and uses a mixture of mathematics and the graphical languages of schematic and block diagrams. Descriptions based on such models may be refined by replacing functional blocks by the diagrams of circuits which realize them. However, the designer may find it necessary to use a frequency domain model to perform some analysis or to use some thermodynamic model to evaluate the physical feasibility of a design. Since different aspects of the system being designed can be most easily analyzed in terms of different models, techniques for switching from one to another must be well defined.

An important observation is that there is a central design model, amounting to an abstraction of the physical system, in terms of which the growing design is expressed. Excursions may be made into other models, but the results are always related back to

the design model. Therefore, a requirement on a design model is that it contain the information necessary to define the mappings between it and each of the other models used (Figure 2). An implication of this is that a central model is never finished; as the state of the art and knowledge change, it must be changed in response. In this sense, the design model can be considered the framework of the whole design process.

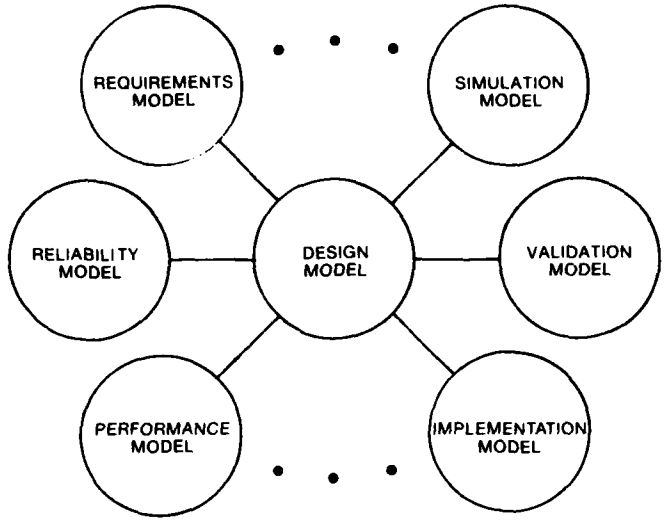


Figure 2. Models for Design, Analysis, and Realization

Multiple models, then, are required for the process of designing an information processing system. Indeed, queueing networks for performance analysis, Petri nets for concurrency description, state transition systems for program validation, initial algebras for data type specification, denotational semantics for programming language definition, and simulation models for analysis of operational behavior are all various models currently in use.

Scope of a Design Model for Concurrent Systems -- Since a design model is a framework within which people can think about a system, its associated language will be used by the various people involved with the development of a given system. Hence, the model must encompass at least the system characteristics outlined in 1.2. It is apparent that several fundamental notions frequently recur in these system characteristics:

- Information Structure - There is the notion of data as a repository of information, along with the notion of operations which affect the information content of data.

The information in a datum exhibits some structure, and pieces of information have certain properties and satisfy certain relationships. The data together with their associated operations are referred to as the information structure.

- Control Structure - There is also the notion of an activity as the behavior over time of some agent at a well-defined location within the system. The pattern of behavior of an activity exhibits some structure, ordering subactivities in time; this structure is often referred to as the control structure of the activity. Both order in time and place in time are fundamental notions.

- Topological Structure - There is the notion of communication among activities; they exchange data values with one another and with the system's environment over well-defined communication paths. These paths define the communication topology of the system. The exchanges define the ways activities may interact with one another. These exchanges may follow some set of rules for orderly communication. Activities may have to know one another by name in order to establish communication. The paths, interaction rules and naming rules together may be referred to as the topological structure of the system.

With these notions in mind we examine system characteristics. The global pattern of what information exists, where it is, how it is accessed and by what, all concern data, its operations, what sites contain it, and what activities use it. The requirements on connections between nodes involve communication paths, data structure and properties, communication among activities, behavior in time, and relationships between the activities and the data they use. The individual nodes involve the data and activity structure of sites and their behavior in time, while node interaction concerns the properties of communication paths. The operational properties of robustness, security, fault-tolerance, reliability, performance, correctness, and viability are all related to the dynamic behavior of activities in response to actions and changes both from within and from without the system.

2.2 NOTATION

A language for describing concurrent systems must be able to describe systems at the user requirements level, the detailed design level, and all levels in between. The

information presented at any level includes functional, operational, and performance properties. There are specific requirements on the language which must be satisfied in order to be able to express these properties. Additionally, there are certain properties which a language for describing systems must enjoy for it to be usable in the design process.

2.2.1 The Design Process

During the design process, a complete description of all the component parts of a product and how they are assembled is prepared. It must be possible to demonstrate that the product as described satisfies the requirements. For complex systems, this demonstration must be both more complete and more economical than experimentation on the finished product.

A design description of a system is an abstraction of that system; it is not the system being considered. Rather, it contains a complete, consistent, unambiguous system characterization suitable for a particular purpose. Implicit in this definition is the assumption that different descriptions of the same system may be appropriate for different purposes. During synthesis it is necessary to describe the system being realized without constructing it. During analysis one must be able to describe the system in such a way that information may be extracted or derived from the description.

A critical requirement in design activity is the ability to utilize feedback. Feedback results from the analysis of information derived during various design stages. The designer uses feedback to determine whether choices made in one design step are correct or incorrect with respect to the requirements. Correctness arguments provide assurance that the design step just taken is in fact a correct and desirable one, while incorrectness demonstrations show that the design step just taken was the wrong one and, it is hoped, will provide some insight into making a more appropriate choice.

2.2.2 The Definition Language

Characterizations of systems are based on one of the multiple models mentioned in 2.1.2 and are stated in the language associated with the appropriate model. For

example, the languages associated with the various models are mathematics and diagrams for queueing networks; places, transitions, and tokens for Petri nets; mathematics and symbolic logic for state transition models; mathematics for initial algebras and denotational semantics; and simulation languages for simulation models.

The language associated with the design model is referred to as the definition language. The relationship of the definition language to the languages asociated with the various analysis and realization/implementation models is portrayed in Figure 3. Obviously, the relationships among the various languages is analogous to the relationships among the associated models in Figure 2.

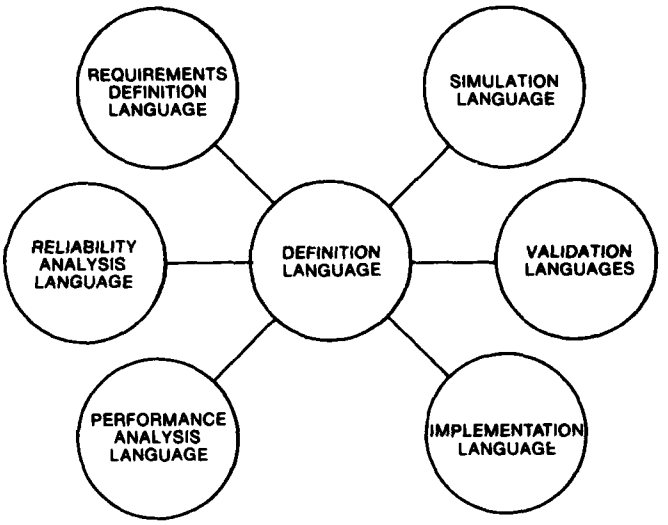


Figure 3. Languages for Design, Analysis, and Realization

The definition language is the language for expressing a characterization of a system, i.e., the language for defining a system in terms of the concepts of the design model. It must provide means for:

- Specifying requirements for a system
- Describing the design of a system

The nature of these two components of a system definition suggests that the definition language be composed of two constituent languages:

- The specification language

- The description language

The CSDL effort is primarily concerned with the development of a description language associated with a design model for concurrent systems. However, when the use of CSDL in the design process is considered, both constituent languages of the definition language are required to generate a complete, consistent, unambiguous characterization of a concurrent system. Therefore, a discussion of both a description language and a specification language are included in the presentation of CSDL in this report. Consequently, issues involved in both specifications and descriptions are reviewed here.

Specifications -- The software engineering literature abounds with discussions of specifications. There appears, however, to be no consensus on the definition of the term. Phrases like "requirements specifications," "design specifications," and "implementation specifications" only add to the confusion.

The terms "specifications" and "system specifications" will be used to mean "the precise definition, in some language associated with an appropriate model, of what the system component under discussion actually does." The component may be at any level, from the entire system to a single primitive, and it may be function, behavior, operational properties, or performance which is being specified.

Specifications are used to document the behavior of a system and its components both for the user of the system and for its designers, builders, and maintainers. They are also used by those analyzing the system for purposes of comparison with other systems. Since it is generally necessary to be able to show that a specification satisfies the given requirements, all aspects of a system subject to requirements must be specifiable; this means that specifications will in general use the full scope of the underlying model. Also, since specifications and requirements must often be compared, the language used for expressing requirements and specifications should be very similar; for practical purposes, they should be the same language.

If "requirements" are defined to be "what the component is supposed to do," then design verification could be defined to be the creation of a formal argument that the

specifications "satisfy" the requirements; that is, any component which behaves as described in the specification will behave as described in the requirements. This relationship is indicated by the "=>" symbol in Figure 4. It also follows that when a hierarchical development technique is used, the specification of level i becomes a requirements statement for level i+1 as shown in Figure 4.

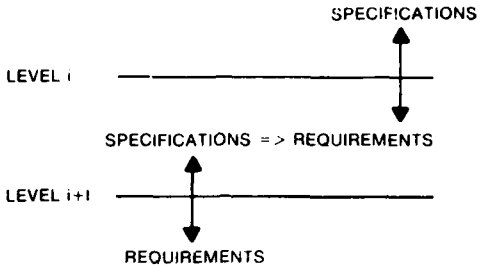


Figure 4. Levels of Requirements and Specifications

Descriptions -- The terms "description" and "system description" will be used to mean "the precise definition, in some language associated with an appropriate model, of how the system component under discussion actually behaves and is structured." The component may be at any level, from the entire system to a single primitive, and it may be function, behavior, structure, or performance which is being described.

A description identifies (in a declarative sense) the system components and their interconnections. Additionally, a system description exhibits (in a procedural sense) the dynamic behavior of the system components and their interactions. In the design process, it becomes necessary to provide correctness arguments which establish the fact that the described system actually behaves as required by the specifications. Therefore, the description language should lend itself to the evaluation of a system description against specifications formulated in the specification language.

2.3 SYSTEM ANALYSIS

In this subsection, the activities involved with the analysis of system designs are discussed, and problems to be addressed by analysis techniques and strategies are identified.

2.3.1 Analysis of System Designs

System design produces a definition of a system expressed in terms of abstract components. This representation constitutes a blueprint for the realization of a system in terms of real components. System requirements must be met in order for the system to perform as desired. The system requirements can be met by the realized system only if the system design, from which the realization is derived, meets the same requirements.

Two levels of analysis can be identified:

- The analysis of a system design involves understanding and evaluating the functional and operational behavior of a system design, and demonstrating that the system design satisfies the system requirements.

- The analysis of a system realization involves observing the functional and operational behavior of a system realization and demonstrating that the system satisfies the system requirements.

As far as system analysis is concerned, the CSDL effort is primarily concerned with the definition of a formalism supporting performance analysis of system designs. The CSDL approach to performance analysis is presented in Section 6.

To incorporate the analysis of system properties into a structured design process, analysis needs to be carried out in each design step. Therefore, system properties must be expressed at each level of design so that the original, global system requirements can be related to the appropriate individual system components which are derived during design. To show that a system design has the desired properties requires a demonstration that each system component meets its requirements and that together they satisfy the global system requirements.

For an analysis technique to be applicable, system requirements need to be derived and expressed in terms of concepts of the model underlying the analysis. The selection of such a model depends on the level of abstraction used in the associated design step, the nature of the system property to be demonstrated, and the type of analysis to be

performed. Therefore, along with the refinement of the design, the requirements need to be elaborated and their representation in the design model (see 2.1.2) needs to be mapped into a model which is attuned to the intended analysis activity.

2.3.2 Analyzing Properties of System Designs

System properties are constrained by the requirements placed on the functional and operational behavior of a system. Thus, the basic purpose of analysis is to

- Derive statements about the properties of system components
- Demonstrate that these properties satisfy the requirements for the system components
- Combine properties of system components, and demonstrate that the combined system properties satisfy higher level system requirements

The definition of the fundamental system behavior is based on the design model. Nevertheless, alternate models need to be employed in order to carry out particular analyses. Models other than the design model may be needed to obtain qualitative or quantitative measures of system behavior. A parameterized set of designs may be evaluated in this manner; parameter variation then allows alternative candidate designs to be analyzed.

In each design step, a distinction between requirements and specifications can be made (see Figure 4):

- Requirements of a system define the characteristics which any design of the system must possess.
- Specifications of a particular system design define the characteristics which it in particular possesses.

Clearly it must be demonstrable that a system's specifications assure that its requirements are satisfied. Hence, during the design process the requirements are

used to aid in the construction of a system design whose specifications satisfy the requirements; this is the principle of constructiveness. In particular, parameterized component requirements can be used to describe a class of designs, with analysis used to determine component parameter values such that the overall design meets the system requirements. An important implication of this is that both requirement and specification parameters must represent a particular system characteristic in the same terms.

Analysis techniques may evaluate the consistency of system designs with given requirements either statically or dynamically.

Static Analysis -- Static analysis techniques evaluate system properties by examining the relationships among various system parameters. Consistency of parameters between levels of system definition is determined without having to consider the operational behavior of the system. Examples include analyses and inferences from state transitions defined in terms of the definition language for performance, reliability, security or program verification.

Figure 5 illustrates several kinds of static analyses that can be performed on the basis of a system definition. Program, security, reliability, and performance verification of a system definition can be obtained by direct analysis of all or part of a representation of that system in terms of the definition language. Topological profile analyses also directly operate on a system definition. They can be performed by appropriate definition language interpreters rather than more sophiscated analysis techniques. Implementations of these techniques or interpreters accept definition language representations as input and then derive characterizations of system properties.

Dynamic Analysis -- Dynamic analysis techniques evaluate system properties by observing the operational behavior of a system. Separate analyses of the operational behavior at several levels of system definition may provide information required for the dynamic analysis of consistency between these levels. An example is the representation of a system in the form of queueing networks or simulation programs for performance analyses.

Figure 6 illustrates several kinds of dynamic analyses that can be performed on a CSDL system definition. A representation on the basis of performance, reliability, or

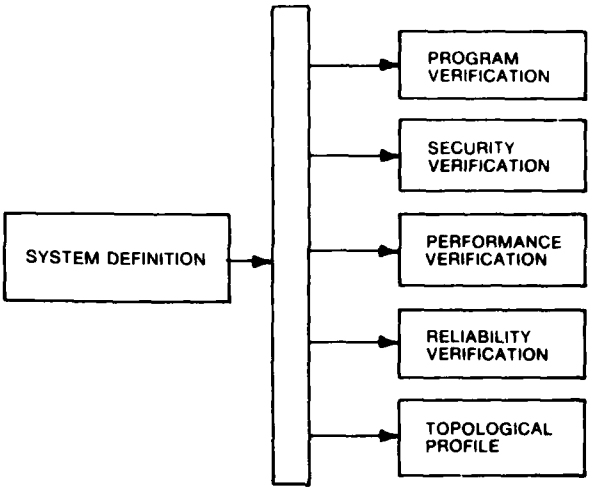


Figure 5. Static Analysis

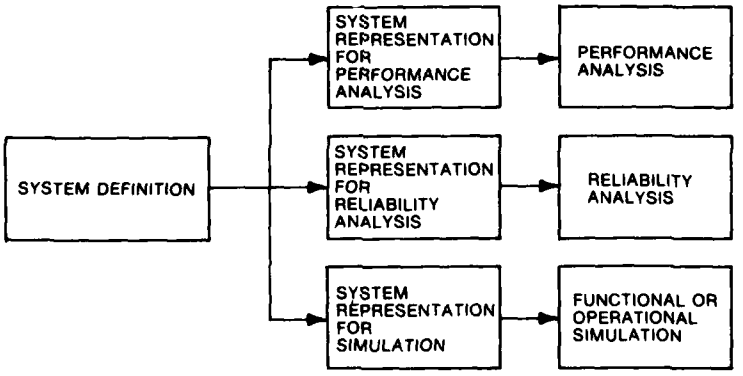


Figure 6. Dynamic Analysis

simulation models is first derived by mapping relevant information contained in a system definition in terms of the definition language onto the appropriate analysis model. An analysis of the derived representation is performed, and the characteristics of the corresponding system property is obtained.

2.4 SUMMARY

In conclusion, methodologies and languages for system design must address at least the following issues concerning the analysis of system properties:

- The derivation and verification of system properties from system definitions
- The expression of system properties at all levels of design
- The mapping of definitions of system properties among levels of design
- The mapping of definitions of system properties among different models
- The application of analysis techniques associated with system representations based on appropriate analysis models.

SECTION 3
CSDL CONCEPTUAL FOUNDATIONS

3.1 GENERAL APPROACH

In this subsection the general viewpoint of the design of CSDL and for the use of CSDL in concurrent system definition is presented, and the basis for this viewpoint is established.

3.1.1 Basis in a Model

A major portion of the development of CSDL is the definition of a precise computational model for concurrent systems. There are several reasons for putting this emphasis on the theoretical underpinnings of CSDL.

First, the semantics of the various language constructs can be defined in terms of the model. As a result, any doubts about the meaning of some expression in CSDL may be resolved by checking the semantic definition and reviewing the definition of the model. In other words, the availability of a model makes unambiguous communication possible.

Second, basing the semantics of CSDL on an explicit, predefined model makes it possible to develop a single coherent view of concurrent systems. The essential concepts of the model are (it is hoped) few in number, and all other notions are defined in terms of them. The basic notions in the model become the terms in which a user of CSDL will think when designing a system. Given an understanding of the model, CSDL becomes an effective tool for expressing one's thoughts since the constructs of the language correspond to the way one is thinking.

Third, the existence of a defined model aids the language development process by encouraging careful consideration of meaning when introducing constructs. The technique for language development used here is to define linguistic constructs which correspond to basic notions in the model and to establish grammatical rules for building up more complicated utterances from simpler ones. An additional possible

benefit of this technique is that the size of CSDL can be kept under control if the model can be kept simple.

Finally, the existence of the model aids the language development process by providing a compendium of assumptions made about concurrent systems. The model embodies an attempt to define precisely the "essential" properties of concurrent systems while ignoring the "irrelevant" ones. When attempts to use the language indicate that some important property of a real system cannot be expressed, the model is scrutinized to determine whether it supports the required concepts. If it supports them adequately then the language can be modified to make them more expressible. If it does so poorly or not at all, extensions to the model can be made to handle the additional information. This extension process makes it possible to enhance the utility of the language without having to redefine it.

In summary, the definition of a precise computational model enables the orderly development of a coherent, unambiguous language for defining concurrent systems.

3.1.2 System Architecture

One very common approach to describing concurrent systems is to consider concurrency as another control construct, like iteration and selection; this will be called the concurrent programming approach. A new block delimiter pair indicating concurrent execution, such as "cobegin...coend", is introduced, and each program which will run concurrently with another has a special declaration, like "process" or "task". In addition, a mechanism for enforcing a form of synchronization is also present. Mutually exclusive access to data may be enforced by means of monitors. Mutually exclusive execution of procedures can be enforced via critical regions. Delays may be enforced with semaphores or eventcounts. Finally, synchrony may be achieved through rendezvous or blocking-send/blocking-receive protocols for communication.

The approach used in developing CSDL, however, views concurrency as a relationship among activities; this will be called the architectural approach. A concurrent system is viewed as a structure made up of a number of components of several different sorts. The manner in which these components are put together defines the architecture of the system. The details of the architectural view of CSDL are presented in 3.2.

There are several reasons why the architectural approach is to be preferred to the concurrent programming approach for our purposes. First, CSDL is intended to define the structure and organization of information processing systems. As such, it will be used to express the means by which various capabilities are supplied to a concurrent programming environment. Thus, for example, Concurrent Pascal [BRIN73] provides monitors simply by declaring them; CSDL would be used for describing how they are built. Again, Ada [HONE80] presupposes the rendezvous mechanism, but CSDL would be used to define the way in which the rendezvous synchronization is to be provided, along with the specification of the scheduling policies desired. Finally, Hoare's CSP [HOAR78] presumes a blocking send and receive communication protocol; CSDL would be used to describe how this would be implemented over, for example, a bus communication system. The point here is that the intended uses of a concurrent programming language and CSDL are somewhat complementary -- the programming language is used to express how an application is built on some synchronization and communication mechanisms; CSDL is used to describe how these mechanisms are built from given resources.

Another reason for using the architectural approach is to allow flexibility in choosing concurrency mechanisms for a particular problem. Each such mechanism is suited for a certain class of problems but is poor for other problems. The architectural approach makes it possible to select whichever mechanisms are best suited for a given problem by allowing the designer to specify, and later build if necessary, the synchronization and communication properties of any component.

The most important reason for using the architectural approach, however, is that CSDL is used to express a system definition during the design process. As such, it must allow the expression of a system definition which is not biased towards a software, firmware, or hardware realization. This re-emphasizes the design issue. In our opinion a program, be it in Ada or assembly language, constitutes the description of an implementation. This description is incomplete without a definition of the system implemented by the program. CSDL is the language for that definition.

3.2 DESCRIPTION AND SPECIFICATION OF ARCHITECTURE

The architecture of a system encompasses both the components that make up the system and the way those components are connected. The declaration of these components and their connections comprises a description of the system. Each component of a system behaves in a certain way and contributes to the behavior of the entire system. Hence, there is associated with the entire system and with each of its components a specification of its behavior. The kinds of components and connections that can make up a concurrent system are introduced and discussed in 3.2.1, and the sorts of things that must be specified about them are discussed in 3.2.2.

3.2.1 Description of Architecture

In the view taken for CSDL, three kinds of components are recognized -- data, programs, and modules. These components, together with their interrelationships, constitute the information structure, the control structure, and the topological structure of a system architecture (cf. 2.1.2).

Data -- Data objects are repositories of information used by a system. Some objects are permanent in the sense that they last as long as the system does, though their contents may change over time; an example would be a data base. Other data objects are transient, coming into existence for some temporary purpose of the system and then vanishing. An example would be local variables allocated upon entry to a procedure and de-allocated upon return. The distinction between permanent and transient data has an important bearing on resource allocation and memory management strategies.

Another important characterization of data concerns an object's behavior from the point of view of the module which contains it. This characterization determines the scope of information required to understand the module's behavior.

A data object is said to be passive if its state changes only as a result of operations performed on it by the module which contains it. Typically the variables of conventional programming languages are understood to be passive. In fact, erroneous behavior of a "working" or "correct" program is often due to presumably passive data being altered by an unreliable run-time environment.

A data object is said to be active if it is not passive, that is, if its state can change without the containing module performing an operation on it. The shared variables and communication interfaces of some programming languages are examples of active data. Also, memory-mapped I/O locations, program status words, and I/O ports are hardware data objects which appear active to a particular module.

The importance of the distinction between active and passive data lies in the scope of information required to understand the behavior of a module. If a module contains only passive data then its behavior depends only on the initial state of its data and the programs executed within it. If it contains active data then its behavior depends also on the behavior of its environment, including any other modules which can cause changes to its active data. It is clearly crucial that the active data of a module be known explicitly; in general, if some active object is presumed passive, no understanding of the module's behavior is possible.

The information contained in a data object at some instant is referred to as its value. There is generally a predefined set of things which can be done to a data object; that is, there is certain information which may be extracted, and there are certain ways the value may be changed. This is called the set of operations which may be performed on the data object. The set of all possible values together with the set of operations is called the object's data type.

An interface object is a collection of active data objects used to mediate communication among modules. The elements of an interface are divided among several modules, and no one element may be found in two different modules. Certain of these elements act as transmitters by having data put into them by the containing modules, and the others act as receivers by having data extracted from them. The specification of an interface type determines the connection topology of the transmitter and receiver elements.

Programs -- A program is a sequence of operations applied to members of a set of data objects. A program is described using some algorithmic notation, for example a sequential programming language. The effect of a program's execution is defined in terms of the set of data objects referenced and the objects altered. In addition, a program may have local data objects which exist only during a particular execution of

the program. A program may effect a particular change in value of some of the permanent data, or compute some function of it; such a program must terminate on valid inputs.

In concurrent systems there frequently are programs whose functionality is defined not by a relation between an initial state and a terminal state, but rather by behavior during the period of execution. Such behavior is typically defined by specifying how the program changes state and sends data to its environment in response to receiving a stimulus when in a certain state.

Modules -- A module defines a collection of data objects, and no data object is entirely contained in two different modules. There are two forms of module, one made up of a set of sequential programs manipulating the data, and one made up of smaller modules connected by interface objects.

In the first form the programs form a hierarchy with one root. This hierarchy is the program calling tree. The single root of the hierarchy is a sequential program called the controller. The controller has the direct responsibility of ensuring that the module accomplishes what is required of it. The objects consist of the permanent data for the programs together with elements of interface objects, as described below.

In the second form of module the data objects are divided among several modules. There exist interfaces connecting the modules over which information is transferred. The modules, communicating with one another over interfaces, together ensure that the containing module accomplishes its objectives. Notice that a single program, a set of programs, and a concurrent system can each be thought of as making up a single module; the differences between them show up in the structure of the module.

In summary, then, a concurrent system is viewed as a collection of modules communicating with one another over interfaces. Each module is made up of a set of data objects and programs which manipulate them in response to requests from other modules or from the environment and which send values out to other modules or the environment.

3.2.2 Behavior Specification

The essential properties of each data object are defined in its type specification. The type specification defines the set of values the object may assume and what its initial value may be. For each operation there is a specification of what value it returns, and of how it changes the value of the object when applied. Active data types, in particular interface types, will have additional behavioral specifications defining dynamic behavior and information passing properties. Within a module specification there may also be specification of relationships among its objects which must be established or preserved by the programs of the module.

For each program in a module which establishes some final state of the permanent data, or computes some function of it, there is a specification of its appropriate input values and of how its final value is related to those input values.

In addition to these static specifications of procedure functionality, there may be behavioral specifications for a module which define how it is supposed to react to incoming information, how output information is related to incoming information and to the values of its permanent data, and what priorities it must follow in determining what to do. These intramodule behavior specifications must be satisfied by the controller of the module.

Finally, properties of information flow in an interface object, dependencies in time of modules among one another, and behavior of the system as a whole are defined as a collection of intermodule behavior specifications for the system.

In conclusion, there are specifications associated with each kind of system component; these include properties of data, functionality of programs, and dynamic behavior of communicating modules.

3.3 DESCRIPTION AND SPECIFICATION IN CSDL

There are mechanisms in CSDL by which the description and specification of a system and its components may be expressed. In this subsection these mechanisms are

introduced, and their association with the architectural terms defined in 3.2 is presented. This subsection is an informal guide; the model for the architectural view is presented in 4.1, and the constructs of CSDL, along with their definition in terms of the model, are presented in 4.2. Table 1 shows the correspondence between the system architecture notion of 3.2 and the CSDL mechanisms described here.

3.3.1 Description of Components

Data -- Data objects are described through name-type associations within a module. Type definitions contain specifications of the set of possible values, the initial state, and the available operations for the type. An abstract data type is defined by associating a type name with a declaration of the type. This declaration expresses the way an instance of the type is to be thought of as a conceptual object space. It is important to bear in mind that this structure generally does not resemble the way in which an instance of the type will actually be represented; the representation is defined in a separate refining machine. This information structure, together with a specification of a type invarient which restricts the structure in some way, implicitly defines the set of values which instances of the type may take on. There may also be a specification of the initial value which an instance will have upon its creation.

Each operation is declared as a name, possibly accompanied by a list of formal value parameters; if the operation returns a value, then the type of the returned value is also declared.

It is also possible for a type to be active; in this case the type definition will also contain behavioral specifications of spontaneous state changes. For example, inlets and outlets are active to the machine which contains them. The type definition of a channel includes behavioral definitions of temporal relationships between state changes at one end and corresponding changes at the other.

The permanent data of a module are declared outside the scope of any program, and transient objects are declared in the scope of the programs to which they are local. Interface objects are declared as object names associated with inlet or outlet types as appropriate; inlet and outlet types are discussed below. Interface type declarations and object descriptions appear in the module which contains the communicating elements.

Table 1. Architectural View and CSDL Correspondence

Architectural View	CSDL Mechanisms
Modules Specification o Activities o Information flow among modules o Cooperation rules o Ordering o Initial state Description - - o System, submodules, interconnections o Process, data objects, procedures, controller	Machines Specification - - Temporal and functional assertions Description - - o Partitioned machine: partitioning of data objects, channels o Simple machine: permanent data, programs, controller
Programs Specification - - o Behavior over time effected by controller o Initial conditions o Final result of terminating sequential programs Description - - Sequential programs	Programs Specification - - o Behavior: actions, temporal and functional assertions o Static: input constraints, input/output relation Description - - Algorithmic design language based on Dijkstra's guarded commands
Data Specification - - o Set of values o Operations o Relationships among objects Description - - Permanent data and local variable declarations	Objects Specification - - o Types: abstract description of values initial state, type invariant, operations with input constraint, input/output relation o Invariants: relationships among objects to be established or preserved by programs Description - - Declaration of binding of names to objects

Programs -- Each program is declared as a program name, possibly accompanied by a list of formal value parameters, with a program body. The body contains the declaration of all local variables and the program text. The text is made up of statements in an algorithmic design language based on Dijkstra's guarded commands [DIJK76]. The design language has constructs for serial concatenation, repetition, selection, and procedure call.

Machines -- The major unit of definition in CSDL is the machine [BOYD78a]. A machine in CSDL defines a logical functional unit of a system; it corresponds to the architectural notion of a module introduced in 3.2.1.

There are two possible structures for a machine, simple and partitioned. A simple machine contains a hierarchy of programs rooted in a controller. A partitioned machine is made up of several machines communicating over an interface object. Any permanent data object will be found in exactly one of the partition machines.

Channels -- A standard interface type is the channel, used to link two modules together. A channel has two "ends" -- an object of type inlet and an object of type outlet; these are similar to the windows of the HXDP executive [BOEB78]. The outlet lies in the space of the sending machine, and the inlet lies in the space of the receiving machine. The data type of the information which goes across the link is specified along with the declaration of the channel object.

Two operations may be applied to the outlet end of a channel: put a value of the type associated with the channel, and check to see if the last value sent has gone. The reason for the check is that the put operation does not block, so that there may be a delay between the time a new value is posted and the time it is transmitted. If multiple puts are performed during the delay time, the last value posted by the time the delay period has elapsed is the one transmitted.

Also, two operations may be applied to the inlet end of a channel: get a value of the channel type, and check to see if another value came. There is no buffering, so later arrivals may overwrite earlier ones. If multiple gets are performed before the next arrival of a new value, the value gotten is the last one to have arrived.

3.3.2 Behavior Specification

Static Specifications -- Static specifications include initial states, function specifications, and invarients as described in the following paragraphs.

Initial States The initial state of the objects of a machine or of a data type's conceptual object space is specified by an assertion which defines the desired relationship among the objects. The intended interpretation is that, when the machine or type instance comes into existence, its objects are guaranteed to satisfy the assertion.

Function Specifications -- In order to discuss the specification of programs and type operations, the term objects will be used to refer to either a program's permanent data or a type's conceptual object space, and function will refer to either a program or a type operation. The conditions which must be satisfied for a function to be properly invoked are specified by an assertion which defines the required relationship among the objects and the function's parameters. The effects of a function are specified by an assertion which defines the desired relationship between the state of the objects when the function terminates and the state of the objects and parameters when the function is invoked. If the function returns a value then the assertion also defines the desired relationship between the returned value and the state of the objects and parameters when the function is invoked. Thus, the specification of a function defines a relationship between input and output states together with the constraints on the input. The intended interpretation is that invocation of the function when the objects and parameters satisfy the input constraint is guaranteed to terminate with the objects in a state correctly related to the input state.

Invariants -- An important kind of static specification which relates data to computations is the invariant. An invariant is a relation among data objects which, from a point of view external to some scope of control, is preserved by that scope. A type invariant is a property of a type's conceptual object space established when an instance of the type is created. Furthermore, each type operation must have the property that, if an instance satisfies the invariant and the operation is invoked so that its input constraint is satisfied, the instance will satisfy the invariant when the operation terminates.

Similarly, a loop invariant is an assertion about the data of a program which is established before entering a repetitive construct and for which it can be shown that if the assertion is satisfied by the data prior to execution of the body of the construct then execution of the body will leave the data in a state which satisfies the assertion.

Finally, a data invariant is an assertion about the permanent data of a machine which is established by an initialization procedure when the machine comes into existence and which if it is satisfied when the program is invoked, every terminating program of the machine will ensure is satisfied upon termination. Notice that a data invariant of a machine may very well constitute a loop invariant of that machine's controller (indeed, the controller may have been designed with this in mind) and that the data invariant of a machine which refines an abstract data type is a translated form of the type invariant of the type which it refines.

Behavior Specifications -- The behavior of a machine or active data object over time is specified by restricting the set of all possible behaviors to those which satisfy prescribed constraints. The basic behavioral concept is that of the event, which is defined to be the change in value of some data objects. Behavior is specified by prescribing events which must occur as the result of other events, by restricting events to occur in certain orders, and by defining properties which are satisfied by data objects when certain events occur.

An event is specified by an assertion which defines a constraint on the initial state and the desired relation between the final state and the initial state. This form is similar to that of program and operation specifications, and for this reason event definitions are called actions.

3.4 ORGANIZATION OF CSDL SYSTEM DEFINITION

In this subsection the two basic principles which govern the organization of CSDL documents are presented. Figures 7 and 8 show how the structure of simple and partitioned machine definitions follow these principles. The two organizational principles are that:

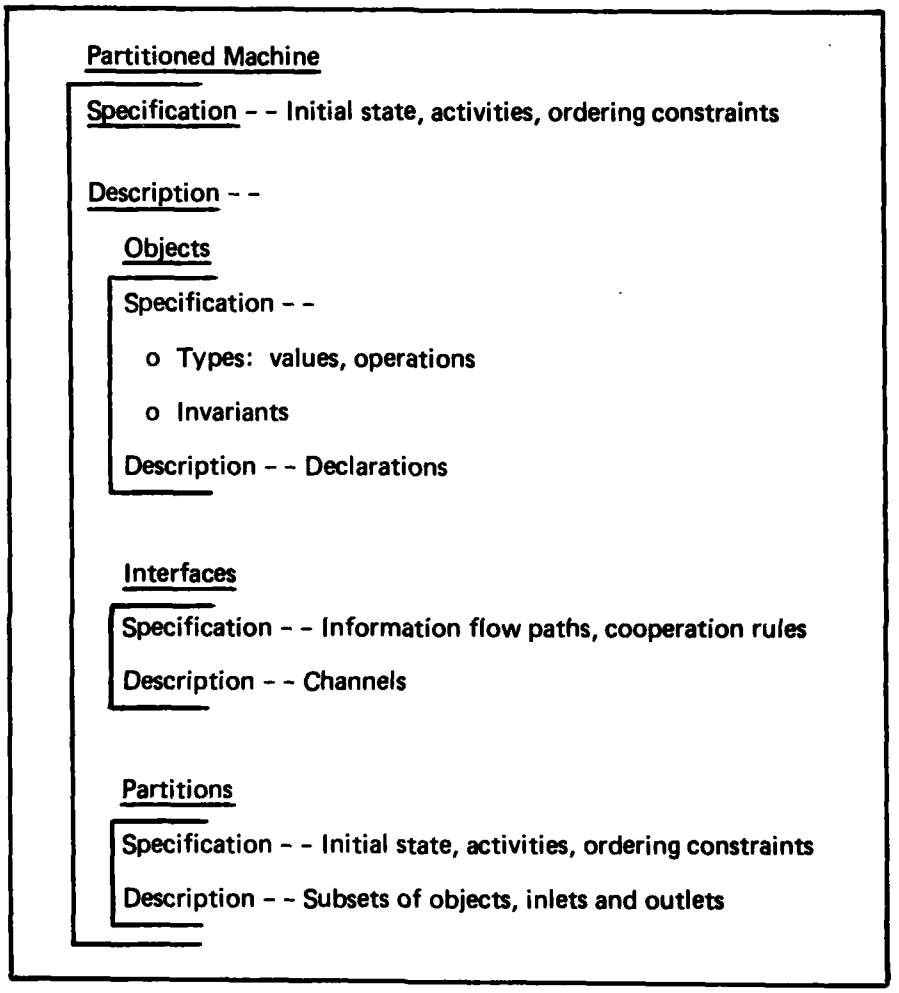


Figure 7. Partitioned Machine Structure

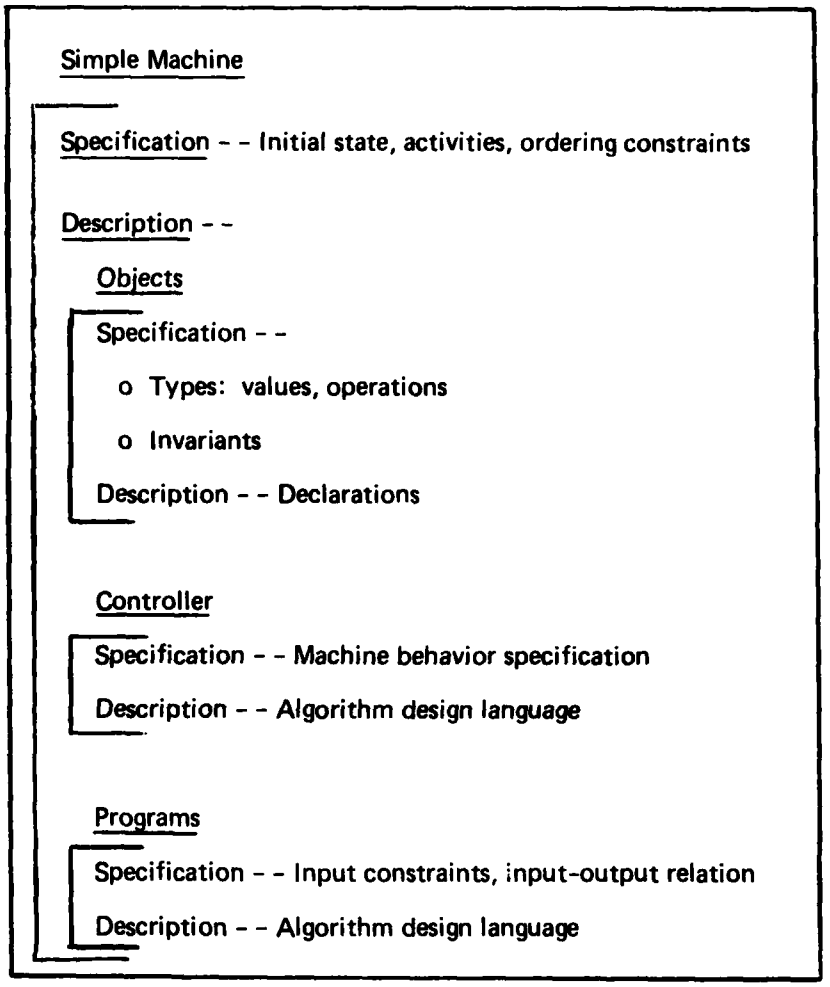


Figure 8. Simple Machine Structure

(1) Both the specification and the description of each defined component must be expressed.

(2) The definition of a system is presented hierarchically.

3.4.1 Specification and Description

The specification and description of each defined component of a system are presented in CSDL as two distinct statements or linguistic fragments. The reasons for this redundancy may be found in the role CSDL plays in the system development process. CSDL is supposed to be a common language by which humans may communicate concerning the design of an information processing system. From this perspective a component specification without a description identifies a design task yet to be accomplished; the only exception is that a component provided by the environment or otherwise already in existence would not have to be designed again. The specification for such a "primitive" component would still have to be given, both to identify the component needed and to enable analysis, validation, and realization of the design.

Similarly, a component description without a corresponding specification amounts to a problem solution without a statement of the problem. If there is no available specification of the component, then how is its place in the whole system to be explained? And if the behavior of the component which contains it depends on the subcomponent's behavior, how can one show that the containing component has been designed properly if the subcomponent's specification is not available? While it may be possible in principle to verify the design by using the subcomponent's description, the amount of design detail present in such a description generally makes such validation infeasible.

The specification of a CSDL machine expresses the initial state, activities, and ordering constraints among those activities. If the machine is simple then its description is made up of the definition of its objects, its controller, and its procedures. If the machine is partitioned then its description is made up of the definition of its objects, logical paths of information flow, the interface between partitions, actual paths of information flow through the interface, and its partitions. Schemas for machine definitions are shown in Figures 7 and 8.

3.4.2 Hierarchical Presentation of System Definition

The number of data objects, interfaces, programs, and machines of any real system is so large that it is impossible to grasp them and their interrelationships as one unit of information. Hence, a CSDL definition of a system is organized so that this information is presented in smaller, more manageable pieces. Each piece is a component, so its specification and description both appear.

The basic structural scheme of CSDL is the hierarchy, in which a component is defined in terms of subcomponents which are in turn defined until certain "primitive" components are specified but not further described. CSDL definitions are based on three hierarchical schemes corresponding to the three different component types: machine, program, and data.

Machine Hierarchy -- A partitioned machine is the root of a hierarchy of machines, the intermediate members of which are themselves partitioned machines and the terminal members of which are simple machines; such a hierarchy may sometimes be informally referred to as a system for purposes of discussion. The partitions of a machine are called components.

The definition of an intermediate member of such a system contains definitions of its objects, how the objects are divided up into partitions, and the interfaces which connect the partitions. Each named partition is associated with a machine name; the decision whether to make a component simple or partitioned is deferred until its machine is defined. Thus, a number of similar components may be declared which differ only in the objects which they contain; these components are defined by one machine. This separation allows the use of a component in a containing machine to be considered independently of how it is itself defined. The definition of a terminal machine of a hierarchy contains definitions of its objects, the procedures, and the controller program. In Figure 9 the machine hierarchy is represented by nested boxes. The outermost box, representing the top level machine, M0, has data objects a1, a2, b1, b2, c1, d1, d2, and e. They have been partitioned into two machines, M11 containing a1, a2, b1, b2, c1, and M12 containing d1, d2, and e. M11 and M12 are connected by channels c11 from M12 to M11 and c12 from M11 to M12. M11 is in turn partitioned into three simple machines M111, M112, and M113, with channels c111

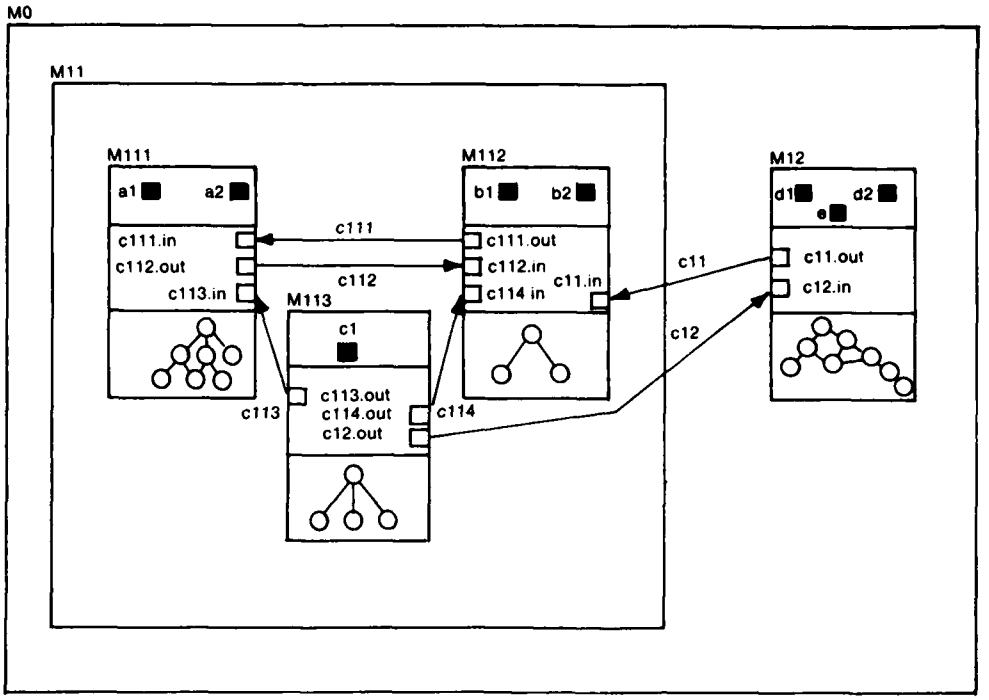


Figure 9. Machine and Program Hierarchies

from M112 to M111, c112 from M111 to M112, c113 from M113 to M111, and c114 from M113 to M112.

Program Hierarchy -- Within a simple machine the procedures form a hierarchical structure (strictly speaking, a directed acyclic graph since a procedure may be invoked by more than one other program) with the controller program at the top. This hierarchy is derived by procedural decomposition. In Figure 9 the simple machines M111, M112, M113 and M12 each have program hierarchies, indicated in the figure by trees; in each case the single root node corresponds to the controller for the machine. Note also that the inlet and outlet ends of all channels become permanent data objects in simple machines.

Type Hierarchy -- The decisions as to how data types are represented and their operations are implemented are elaborated incrementally by a type-refinement hierarchy. Each abstract (nonprimitive) data type specified in a machine forms the root of such a hierarchy. A machine is defined whose objects make up the representing data structure, and whose programs contain the implementations of the

type operations. Since the objects of the refining machine may themselves be of some abstract types, a hierarchy is formed in which the immediate successors of a node are the machines containing the first refinement step of each abstract data type used in it.

This refinement hierarchy is based on the data types; conceptually, there is a copy of the hierarchy for each object of the refined type. However, since these copies are identical, there is only one machine document in the system definition hierarchy. In Figure 10(a) machine M0 has a data type "widget" with two objects, "x" and "y", of type widget. Machines M1 and M2 are copies of machine M_widget shown in Figure 10(b); there is a copy for each object. The plane containing machine M0 in the figure is intended to indicate one level of abstraction of data; M1 and M2 are at a lower level. At the level of abstraction of M0 the representation and implementation information defined in M1 and M2 are invisible. In Figure 10(b) machine M_widget contains the representation and implementation information for the type "widget". The representing data structure is made up of the objects "a" of type "Ta" and "b" of type "Tb". M11, a copy of the refining machine for type "Ta", is associated with object "a", and M21, a copy of the refining machine for type "Tb", is associated with object "b". Finally, we see programs 01 and 02 defined on M_widget which implement the operations 01 and 02 declared with the type widget in machine M0.

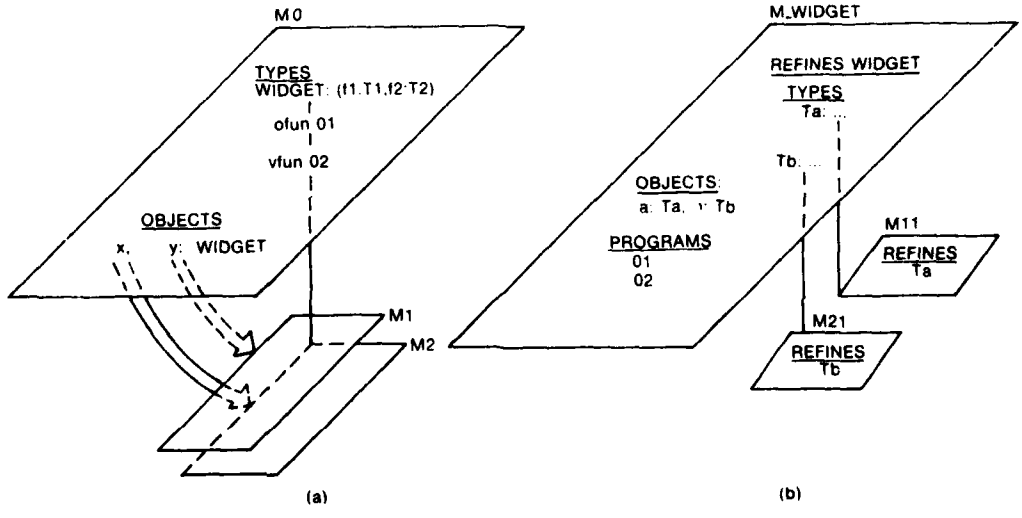


Figure 10. Type Hierarchy

Concurrency and Levels of Abstraction -- It is important to keep in mind that any machine may be partitioned to introduce concurrency. For example, in Figure 10 the refining machine M_widget could have been defined as a partitioned machine. The result is that, at the levels of abstraction at which widget objects are used, only sequential state transitions of the objects are visible; there is no visible concurrency. At the lower level of refinement, however, concurrency shows up as a conscious design decision. This technique can be used for communication refinement in which channel types are refined into processes communicating with a lower-level protocol in order to effect the transmissions used at the upper level.

SECTION 4
CDSL DEFINITION

4.1 CSDL COMPUTATIONAL MODEL

In this subsection a computational model of concurrent systems is developed. This model is based on mathematical set theory, and while simple in content, it provides a unified view of data, conventional sequential programs, and concurrent systems. The model is similar to the relational models of DeBakker [DEBA75] for sequential programs, and Bochmann [BOCH79] for distributed systems. It is also compatible with the abstract model approach to data abstraction [LISK77], which is the approach taken in CSDL.

Subsection 4.1.1 introduces the model for sequential computations, 4.1.2 extends this to concurrent systems, and 4.1.3 is concerned with flow of information in a concurrent system.

4.1.1 Sequential Programs

A variable or data object x is an entity with a name, "x" in this case, which can take on any value V(x) of a certain defined set and upon which any of a defined group of operations may be performed. The set of values together with the group of operations is called the data type of x. The state of x is its value V(x). Given a set X of n objects $x_1, ..., x_n$, where each x_i is of type T_i, the (current) state q of X is the vector of values of the objects; that is

$$q(X) = \langle V(x_1), ..., V(x_n) \rangle$$

The state space S(X) is the set of all possible such vectors.

A single terminating sequential program B defined over a set of objects X effects a state transition on X in that it is invoked with the objects in some state q(X) and terminates with the objects in a state q'(X). Such a program may be modeled as a binary relation M on S(X); M is a collection of ordered pairs of states. The

interpretation of this relation is that the pair $\langle q(X),q'(X)\rangle$ is an element of M just when (1) B is guaranteed to halt when invoked from state q(X), and (2) q'(X) is one of the states in which B can halt when invoked from q(X). From this interpretation it follows that the domain of M (i.e., the set of states which are first components of pairs in M) is exactly the set of initial states from which termination of B is guaranteed. B is said to be deterministic if and only if M is a function, that is, when for each q(X) in the domain of M there is exactly one q'(X) such that the pair $\langle q(X),q'(X)\rangle$ is an element of M.

A particular execution of the program B over X defines a state sequence s(X,B):

$$s(X,B) = q_0(X), \ldots, q_n(X), \ldots,$$

where $q_0(X)$ is the state at invocation of B. If $q_0(X)$ is an element of the domain of M then s(X,B) is finite, and if $q_n(X)$ is the last state in s(X,B) then $\langle q_0(X),q_n(X)\rangle$ is an element of M. Also, for any q(X) not in the domain of M there exists an infinite sequence whose first state is q(X).

Given the state sequence s(X,B) there is an associated state transition sequence h(X,B):

$$h(X,B) = t_1(X), \ldots, t_n(X), \ldots,$$

where the transition $t_i(X)$ is the ordered pair of states $\langle q_{i-1}(X),q_i(X)\rangle$; these transitions are referred to as events. Each event (i.e., state transition) sequence is called a history of the program; the set H(X,B) of all such sequences is the set of possible histories of program B.

4.1.2 Concurrent Programs

The notion of history is introduced in order to describe the effects of several programs in simultaneous execution. A concurrent system is modeled as a collection of m sets of objects $X_1, \ldots, X_m$, with a program B_i defined over each X_i. We impose the further restriction that the X_i's are pairwise disjoint; that is, if $i \neq j$ then the intersection of X_i

and X_j is empty. This is an important part of the model which bears repeating: no object x is a member of two different sets of objects; there is no shared data.

A collection $\{h_1, ..., h_m\}$ of histories, where h_i is an element of $H(X_i,B_i)$, is a system history; the set H of all such collections comprises the set of possible system histories.

Up to this point there is no notion of interaction among the programs of a concurrent system. There is no postulated external or global "clock" history against which to compare progress among different program histories nor is there any concept of interleaving of the various program histories to make a single event sequence for the system history. Even if all the programs terminate, we can say nothing about the time when they stop; we can only talk about the state of affairs of each program upon its termination.

4.1.3 Information Flow

Every interaction between concurrent programs, be it constructive or destructive, involves the transfer of information from one program to another. Programs may cooperate by exchanging information to help one another progress. One program may interfere with another by putting erroneous information, often without the "knowledge" of the victim, into one of its variables.

The transfer of information from program A over object set X to program B over object set Y is modeled by an ordered pair $f_{lk}(A,B) = \langle p_l(X), q_k(Y) \rangle$ of states, where $p_l(X)$ is the l-th state in the history of A and $q_k(Y)$ is the k-th state in the history of B; such pairs are called communication events. The fact that information is transferred is expressed by the property that for some objects x of X and y of Y, V(x) = V(y).

For any system history involving A and B there is a set F of communication events

$$F(A,B) = \{f_{l_1k_1}(A,B), ..., f_{l_pk_p}(A,B), ...\}$$

for $l_1 < l_2 < ...$, and $k_1 < k_2 < ...$. The objects x and y participating in the communication are called interface objects.

There are four events associated with a communication event, two with the source interface object and two with the sink. Let $f_{jk}(A,B) = \langle p_j, q_k \rangle$ be a communication event with source interface object x in X and sink interface object y in Y respectively. We define the events puts, leaves, arrives, and gets on f as

$$\text{puts}(f) = \langle V(x)_{j-1}, V(x)_j \rangle$$
$$\text{leaves}(f) = \langle V(x)_j, V(x)_{j+1} \rangle$$
$$\text{arrives}(f) = \langle V(y)_{k-1}, V(y)_k \rangle$$
$$\text{gets}(f) = \langle V(y)_k, V(y)_{k+1} \rangle$$

Intuitively, the puts event corresponds to loading a mailbox with a message to be sent, and the leaves event corresponds to extraction of the message by the communications medium. Similarly, the arrives event may be thought of as the arrival of a message in a mailbox, and the gets event as its extraction by the recipient.

The histories of each program, together with the events associated with communication, may be used to define a system-wide strict partial order relation precedes. The precedes relation models the notion of order in time. This order relation is defined recursively as follows:

1. Within a history of program B over X, the transition
   $t_j(X) = \langle q_{j-1}(X), q_j(X) \rangle$ precedes the transition
   $t_k(X) = \langle q_{k-1}(X), q_k(X) \rangle$ if and only if $j < k$.

2. If $f_{lk}(A,B) = \langle p_l(A), q_k(B) \rangle$ is a communication event between programs A and B then leaves($f_{lk}(A,B)$) precedes arrives($f_{lk}(A,B)$).

3. If $t_1(X_1)$, $t_2(X_2)$, $t_3(X_3)$ are transitions anywhere in the system such that $t_1(X_1)$ precedes $t_2(X_2)$ and $t_2(X_2)$ precedes $t_3(X_3)$, then $t_1(X_1)$ precedes $t_3(X_3)$.

The precedes relation is partial rather than total since there may be transitions $t_1(X_1)$ and $t_2(X_2)$ such that neither $t_1(X_1)$ precedes $t_2(X_2)$ nor $t_2(X_2)$ precedes $t_1(X_1)$. For

example, suppose A sends data to B, does some other processing, and then receives a reply from B. The events preceding A's puts event also precede B's gets event and subsequent processing. Likewise, the events preceding B's puts event for the reply precede A's activities after getting it. However, nothing can be said about the time relation of B's activities between receiving the data and sending the reply, with respect to A's activities between sending the data and receiving the reply.

4.2 CSDL NOTATION

Given a model for concurrent systems, it is now possible to introduce the CSDL notation and define the meanings of the various constructs. This definition is necessarily informal; while a formal definition, using denotational techniques for example, is possible, it is beyond the scope of this effort. Subsection 4.2.1 explains the description language, and 4.2.2 introduces the specification language fragments used here for purposes of example.

4.2.1 Component Description

Each sort of system component either manipulates or is made up of data objects; hence, the language used for describing the components of a system involves data objects and data values. The syntax for value expressions is first introduced, and then the syntax for describing the various sorts of system components is defined.

4.2.1.1 Value Expressions -- A value may be obtained by referring to some data object or by invoking a program or type operation which returns a value. For example, if "x" is an integer and "com" is a message array then the form

```
com(x)
```

is used to refer to the value at index position x of com; in addition, "x" itself denotes a value. Again, if "sqrt" is a function program which returns the positive square root of a non-negative argument value passed to it, then the form

```
sqrt(3.5)
```

denotes the value which is the positive square root of the value "3.5".

For numeric (real and integer) and boolean values there is a conventional syntax for forming expressions with infix and prefix operators.

Numeric constants may be written as literals denoting integer or decimal fraction values. An integer constant is written as a string of the digits "0" - "9". A fixed point fraction is written as two-digit strings separated by a single decimal point; at least one of the two strings must be made up of at least one digit. A floating-point fraction is written as an integer or fixed point constant immediately followed by the letter "E" and an integer constant. The form "xEy" evaluates to $x * 10^y$. For example

```
3
4.5
3.0E10
```

are respectively integer, fixed point fractional and floating point fractional constants.

Numeric value expressions may be formed with the infix operators "+", "-", "*", "/", div, and "mod", the prefix operators "+" and "-", and grouping operators "(" and ")". The unary prefix operators "+" and "-" have highest precedence, followed next by the binary operators "*", "/", "div" and "mod", while the binary "+" and "-" are lowest. Sequences of equal precedence operators associate to the right. The "mod" operator requires two integer arguments; the form "x mod y" denotes the remainder after dividing "x" by "y". The "div" operator also requires two integer arguments; the form "x div y" denotes the integer quotient of its arguments, which may be either real or integer. Mixed type arguments are allowed for the remaining binary operators; in such cases the values are viewed to be "real". It is important to keep in mind that the numeric types "integer" and "real" correspond to the mathematical integers and reals; they are not finite approximations or machine representations. Thus, every integer value is also a real value. It also follows that the real result of an operation may turn out to be an integer; the use of that result as an integer is then legitimate. Hence the form "4.0 div 2" evaluates to the integer (and hence also real) value 2. For example

```
166/577.2
```

-(x*(3+x*(5))+1)
(i+j) div 2

are valid numeric value expressions.

Boolean constants are "true" and "false". Boolean expressions may be formed with the infix operators "and" (also "&"), "or", and "xor", the prefix operator "not" (also "~"), and the grouping operators "(" and ")". The unary prefix operator not has higher precedence than the binary operators. Sequences of infix operators associate to the right. The operators all have the standard truth-table definitions. For example, in

(i) x or y
(ii) a & b xor c

form (i) evaluates to true if at least one of "x" and "y" evaluate to true, and to false otherwise, while form (ii) evaluates to true if both "a" and exactly one of "b" and "c" evaluate to true, and to false otherwise.

Finally, each relational infix operator "=", "≠", ">", "<", "≥", and "≤" takes two numeric expressions as arguments; the resulting expression evaluates to true or false according to whether the left side value is so related to the right side value. The relational operators are not defined for other than numeric value operands.

4.2.1.2 Data -- The syntax of an object declaration is

ObjectName: TypeExpression

and the meaning is that an object of the type specified by TypeExpression is created, and it is to be accessed by using ObjectName.

Data declared as machine objects come into existence when the containing machine does, and cease to exist when that machine does.

Machine objects are known by name by all programs defined in that machine but by no programs in any other machine.

Data declared as variables within a program are local to that program in the sense that their names are not known to any other program. A program variable comes into existence when the containing program is invoked and ceases to exist when that program terminates.

TypeExpression may denote one of the standard or structured types, or it may be a reference to a declared abstract data type. In the latter case each type parameter must be instantiated to a constant or expression in terms of constants.

Standard Types -- The boolean, mathematical integer, and mathematical real data types are types considered standard in CSDL. Another standard type is the character type "char". The "char" type need not correspond to any particular standard character set, nor is there any implied machine representation or collating sequence. The last standard type is the enumeration of a set of literals. For example, the declaration

```
work_day: (Mon [] Tue [] Wed [] Thu [] Fri)
```

describes the object "work_day" as able to take on any of the symbolic values "Mon", ..., "Fri". These values are not names of other types, objects, or values, nor are they character strings; they name themselves.

Structured Types -- The structured data types of CSDL allow the creation of object structures out of instances of other types. There are six such structures, three for communication and three of general application.

(a) Cartesian product structures define fixed size sets of objects of arbitary types; they correspond to PL/1 structures or Pascal records. The declaration

```
v: (f1: T1, f2: T2)
```

defines v to be a set of two objects, namely f1 of type T1 and f2 of type "T2". Component objects of a cartesian product are accessed by the qualified name or "dot" notation. In our example "v.f1" denotes the object named f1 in v.

(b) Array structures define variable length sequences of objects of some single type. The sequences are indexed by a chain of integer values with no gaps. The declaration

(iii) v: T array

defines v to be a sequence of objects all of type T. Given the array "v" declared in (iii), the following attributes are defined:

"v.lob" denotes the smallest index value.
"v.hib" denotes the largest index value.
"v.dom" denotes the length of the sequence; since there are no gaps in the chain of indices, it is always true that $0 \leq$ v.dom = v.hib-v.iob+1.
"v.low" denotes the object with the smallest index.
"v.high" denotes the object with the largest index.
"v(i)" denotes the object with index equal to i if v.lob$\leq$i$\leq$v.hib; otherwise it is undefined.

Along with these attributes there are a number of operations which may be performed on an array. Given the array "v":

"v.lorem" removes the entry at v.lob and increases v.lob by 1.
"v.hirem" removes the entry at v.hib and decreases "v.hib by 1.
"v.hiext(a)", where a is of type T, adds an object with value a to the high end of v and increases v.hib by 1.
"v.loext(a)", where a is of type T, adds an object with value a to the low end of v and decreases v.lob by 1.
"v.swap(i,j)", where v.lob $\leq$ i,j $\leq$ v.hib, interchanges the values in v(i) and v(j); if the condition is not satisfied the operation aborts (see 4.2.1.3 for a definition of abort).

The form "(k,a1,a2, ..., an)", where k is an integer and a1, ..., an are values of type T, denotes an array constant value. The value "k" is the low index, and the ai's are the elements in order of the sequence; that is, the entry at k is a1, the entry at k+1 is a2, etc., until index k+n-1, whose entry is an.

(c) Discriminated union structures define objects which may take on values of several different types. The declaration

v: (t1: T1 [] t2: T2)

where T1 and T2 are different types, defines "v" to be and object which may be either of type T1 or T2. The identifiers "t1" and t2" are values of the attribute "v.tag". The tag may be examined to discern what type v currently is; it can only be changed by assigning a value of one of the types to v.

(d) Inlet structures define objects to which data arrive from other machines in the system. The declaration

v: T inlet

defines v to be a cartesian product of a boolean flag v.flag and a window variable v.window of type T. The function "v.came" returns the value of the flag, and the operation "v.get" returns the value in the window and sets the flag to false; this is the only value-changing operation available. An inlet is an active object, since a datum arriving from outside its containing machine will set the window to the arriving value and set the flag to true. Only the first get performed after the arrival of a datum will result in a new value being obtained; all gets subsequent to and prior to the next arrival will return the same value as the first one. The two operations and the action of data arrival are indivisible, so data arrival cannot overlap with the invocation of an operation; this constitutes the only guaranteed synchrony constraint. The initial value of the flag on an inlet at the time of its creation is "false". Setting the flag to false by the "v.get" operation corresponds to the communication event "gets(f)", and the setting of the flag to true when a new datum arrives corresponds to the communication event "arrives(f)".

(e) Outlet structures define objects from which data depart to other machines in the system. The declaration

v: T outlet

defines v to be a cartesian product of a boolean flag v.flag and a window variable v.window of type T. The function "v.went" returns the value of the flag, and the

operation "v.put(x)", where x is a value of type T, sets the window to the value of x and sets the flag to false; this is the only value-changing operation available. An outlet is an active object, since at some time after invoking "put" the flag will spontaneously change to true. Data put before this change are lost; only the last value put before the subsequent change to true will be communicated. The two operations and the flag change are indivisible, so the flag change cannot overlap the invocation of an operation; this constitutes the only guaranteed synchrony constraint. The initial value of the flag of an outlet at the time of its creation is "true". Setting the flag to false by the "v.put" operation corresponds to the communication event "puts(f)", and the setting of the flag to true corresponds to the "leaves(f)" communication event.

(f) Channel structures define inlet-outlet pairs. The declaration

v: T channel

defines v to be a cartesian product made up of a T inlet v.in and a T outlet v.out. The inlet field is in the object space of one machine and the outlet field is in the object space of another; the channel is a medium of communication between these two machines. Every value which "leaves" v.out is guaranteed to later "arrive" at v.in, and values arrive in the same order in which they were sent.

Abstract Types -- An abstract data type definition has the form

```
TypeName(Params): (f1:T1, ..., fn:Tn)
let tn: TypeName
init Assertion
invariant Assertion
OpDefs
end TypeName;
```

The parenthesized list of name:type pairs forms a conceptual, abstract object space. The name "tn" introduced after let is a name for a generic instance of the type; it is used as a handle to hang specifications upon. The assertion following init specifies the allowable initial values of any instance of the type; the presumption is that any instance of the type will satisfy the assertion when it is created. The assertion

following invariant specifies a property which is satisfied when the instance is created and which is preserved by any valid application of any operation of the type. That is, it must be demonstrable for each type operation that, if the parameters and the instance's state satisfy the input constraint of the operation and if the instance satisfies the type invariant, then the operation will terminate so that the instance state (and return value if any) satisfies the specification of the operation, and the instance also satisfies the type invariant.

Params is an optional list of pairs of either the form Value:TypeExpression or the form Name:typename. The first form specifies a formal value which is instantiated when an instance is declared. The parameter may appear anywhere that a value may appear, for example, in an assertion. The second form specifies a formal name which instantiates to a known type name when an instance of the type is declared. Formal typename parameters may occur anywhere in the type definition where a type designator is required, for example, in the conceptual object space declarations and in the parameter lists or return clauses of operation declarations.

There are two sorts of OpDefs, operations which alter an instance and functions which return a value without altering an instance.

The OpDef for an operation looks like

```
ofun OpName(Params) returns Value:TypeExpression
  pre Assertion
  post Assertion
```

Params is an optional list of Value:TypeExpression pairs which declare any formal parameters to the operation; they all are values, so no object can be altered by using its name in the actual parameter list of a type operation invocation. The returns clause is optional; it declares the type of a returned value if it is desired for the operation to return a value in addition to altering the instance to which it is applied. The assertion following pre is a precondition which specifies constraints on the parameters and the instance; correct operation is presumed if the precondition is satisfied. The assertion following post is a postcondition which specifies the relationship between the state of the instance at termination and the parameters and

state of the instance at invocation. If there is a return value then the relationship between it and the parameters and state of the instance at invocation is also specified.

The OpDef for a function looks like

vfun OpName(Params) returns Value:TypeName
  pre Assertion
  post Assertion

The returns clause is required, and the parameters are again Value:TypeExpression pairs. The postcondition specifies the relationship between the returned value and the parameter values and instance state at invocation. It is a semantic error for the postcondition to specify an alteration of the instance state; the function is pure. This restriction may be checked syntactically, as discussed in 4.2.2.

4.2.1.3 Programs -- Programs are described using an algorithmic language based on the guarded command set of E. W. Dijkstra. In this language a program may be either a basic statement or a collection of smaller programs combined by certain constructs.

In 4.1.1 a program was said to be modeled by a certain binary relation over an object space. The semantics of the various algorithmic constructs are defined by a semantic function called the weakest precondition predicate transformer. This function may be viewed as taking a program construct and a set of desired output states as arguments and evaluating to the largest set of input states from which termination of the program in one of the given output states is guaranteed. Thus, the value calculated is a subset of the domain of the relation corresponding to the program. Specifically, if B is a program with associated relation M defined over an object space X, and R is a subset of X, then wp(M,R) is a subset of the domain of M with the property that, for any pair <q(X),q'(X)> of M, if q(X) is in wp(M,R), then q'(X) is in R.

Basic statements are made up of the type operation invocation, the procedure invocation, the assignment statement, and the skip statement.

The form of a type operation is

ObjectName.OperationName (Parameters);

if there are no parameters the parentheses are omitted. The parameters are references to values listed in the same order as the formal parameters in the type operation's specification. The only object which may be affected by OperationName is the one named "ObjectName".

The form of a procedure invocation is

ProcedureName (Parameters);

if there are no parameters the parentheses are omitted. The parameters may be references to values or objects as specified in the procedure's specification, and they are listed in the same order as in the specification. Whereas a type operation may only reference or access its parameters and associated object, a procedure may reference or access any object in its containing machine as a global variable in addition to objects and values in its actual parameter list. It is required that the set of global variables referenced be disjoint from the set of objects in its actual parameter list in any invocation. Since there is no notion of procedure nesting, there is no possibility of referencing some procedure's local variables as a global name in another procedure.

The specification of either a type operation or a procedure is made up of an input constraint I and an input/output relation R. The input constraint defines the set of legal input values, and the input/output relation defines a set of ordered pairs <q,q'> of states such that q' bears the desired relationship to q; it is required that I characterize a subset of the domain of R. In terms of the model, the relation M corresponding to a procedure or type operation specified in this way is formed by taking all pairs characterized by R whose first element satisfies I. More precisely, if procedure or type operation Q is specified by input constraint I and input/output relation R, then the relation M corresponding to Q is the set of pairs <q,q'> of states such that both <q,q'> satisfies R and q satisfies I. Given M, the weakest precondition may now be defined naturally as the set of all states q such that both q satisfies the input constraint I, and any pair <q,p> which satisfies R also satisfies S.

The assignment statement, written

(iv) ObjectName := Value

expresses the setting of the value in the object ObjectName to equal Value; clearly this is only defined if the types match. The weakest precondition by which statement (iv) will terminate in a state satisfying R is that Value satisfies R.

The skip statement is the no-operation command; the weakest precondition for it to terminate in a state satisfying R is that the initial state already satisfies R.

Constructs for combining programs to make larger programs are concatenation, selection, and repetition.

In the concatenation of two statements S1 and S2, written

(v) S1;S2

";" is an infix combining operator which denotes the execution of S1 followed by the execution of S2. The weakest precondition by which statement (v) will terminate in a state in a set R is equal to the weakest precondition by which S1 will terminate in an intermediate state which is itself an element of the weakest precondition by which S2 will terminate in R.

The selection of one of several programs based on some condition is expressed by the guarded if construct. If B1, ..., Bn are boolean expressions and S1, ..., Sn are programs, then the statement

(vi) if B1 -> S1 [], ..., [] Bn -> Sn fi

denotes the execution of one of the programs, say Sj, whose associated guard Bj evaluates to true. If more than one guard is true then any one could be selected, while if none are true then the construct does not terminate; this situation is referred to as abort. The weakest precondition by which (vi) will terminate in one of a set of states R is the set of states q such that (a) some guard Bj applied to q evaluates to true, and

(b) for each k such that Bk applied to q evaluates to true, q is in the weakest precondition by which Sk will terminate in R.

The repetitive selection of one of a set of programs based on some condition is expressed by the guarded do construct. If B1, ..., Bn are boolean expressions and S1, ..., Sn are programs, then the statement

(vii) do B1 -> S1 [] ... [] Bn -> Sn od

denotes the repeated execution of (vi) until all guards evaluate to false, at which point the statement terminates. The corresponding semantic function is a rather complicated construction made up of an infinite union of relations; a simpler precondition which is not the weakest is available by application of the invariance theorem. We use "DO" to denote (vii), "IF" to denote DO with do and od replaced by if and fi respectively, and "BB" to denote the set of states q such that some quard Bi, $1 \leq i \leq n$, evaluates to true. If P is a set of states and TF an integer function of the state space which together satisfy the conditions

- Any state q which is in both P and BB is also an element of wp(IF,P)
- Any state q which is in both P and BB also has the property that TF(q) > 0
- Any state q which is in both P and BB, and for which TF(q)=c, is also an element of wp(IF,DEC(c)), where DEC(c) is the set of states r such that TF(r) < c

then P is a subset of wp(DO,TERM), where TERM is the set of state q such that q is in P and q is not in BB.

The first condition says that any one repetition maintains P invariant, the second says that the termination function TF is positive for any state at which is in the invariant and from which another repetition can be made, and the third says that any repetition from a state in P strictly decreases the termination function. The conclusion simply states that under these conditions if the state q at the start of execution of DO is in P then DO is guaranteed to terminate in a state q" which is also in P but which falsifies all the guards.

The usage of this theorem is simply that, if an invariant P and set of guards Bi can be found such that TERM (defined above) is a subset of the desired set R, then P is a subset of wp(DO,R).

4.2.1.4 Machines -- Machines occur in two varieties in CSDL, simple and partitioned. A simple machine is used to define one member of an asynchronous set of sequential computations. A partitioned machine is used to define an asynchronous set of computations. Either variety may be used to define the representation of an abstract data type; in this case the operations will be implemented by programs in the simple machines which make up the innermost partitions. The components of a partitioned machine may themselves be either simple or partitioned. Any machine can be viewed as a structured system component made up of subcomponents which fall into several classes; part of the distinction between simple and partitioned machines appears in the classes of components they each may contain.

Simple machines are made up of a data objects and programs. The data objects and programs are defined as discussed in 4.2.1.2 and 4.2.1.3, respectively.

The set of data objects defined within a simple machine corresponds to one of the sets of objects X discussed in 4.1.2. Some of the objects in a simple machine may be inlets or outlets; these correspond to the interface objects discussed in 4.1.3.

There is always at least one program defined in a simple machine. The program defined first is a distinquished program referred to as the controller for the machine. This program is never explicitly invoked; rather, it starts executing when the machine itself is created. The structure of a controller program is typically a prologue block which initializes the machine objects to some required state, followed by a loop for repeated scanning of inlets. In this loop data arrival is responded to by invoking other programs defined in the machine and sending data out selected outlets. If a controller ever terminates, the machine dies in the sense that there can be no further response to data sent to it from other machines; also passive objects in the machine can no longer change state. In terms of the underlying model, the controller of a machine corresponds to the program B defined over the set X of objects as discussed in 4.1.2.

Since the purpose of the controller is to satisfy the requirements of the machine, its specification will generally be expressed in terms of temporal and communication

behavior. In contrast, the specifications of the other programs in the machine will be expressed in terms of the input constraint and input/output relation discussed in 4.2.1.3.

Partitioned machines are made up of data objects and partitions. The objects are defined as discussed in 4.2.1.2 and 4.2.1.3.

The set of data objects, which may contain inlets and outlets, defines the state of the machine as visible from an external point of view; note that machine requirements are generally inherited from the environment and hence are expressed from such a viewpoint.

The partitions define several disjoint sets of objects; each machine object is allocated to one of the partitions. In addition, channel objects are introduced which connect pairs of partitions to one another. Partitions reflect the design decision to satisfy the machine requirements by parcelling the machine data among the components of a concurrent system, rather than by a single sequential program. The channels define the desired logical communication topology. For example, suppose a machine has declared the objects "db1" and "db2" of type "file". The partition declaration

```
partition
  interfaces
    p1,p2:record channel;

  components
    part1: (db1, p1.in, p2.out);
    part2: (db2, p1.out, p2.in);
```

describes a two-component system, with each component containing a file. The communication topology is (trivially) fully connected.

The interfaces section also contains specifications which define how logical information dependencies specified over the machine objects are mapped onto the communication topology; they also define the flow of information around the subsystem. The components section similarly contains specifications of initial and

final states for each component where applicable along with the functional and temporal behavior of each component.

4.2.2 Specification Language

4.2.2.1 Assertions -- Every specification occurs in the form of a predicate, that is, a statement of a relationship among terms. Static specifications assert relationships among data objects, and behavioral specifications assert relationships among event occurrences and data at event occurrences.

Static Specifications -- In this subsection X will be the set of objects X1, X2, ..., X5. The primitive predicates are the numerical relations "<","$\leq$", ">", "$\geq$" among numerical values, and identity "=" and its negation "$\neq$". The forms

$$X1=3$$
$$0\leq X3\leq X4$$

respectively characterize the set of state vectors <X1, ..., X5> such that (1) the value of X1 equals 3, and (2) the value of X3 is both non-negative and not greater than the value of X4. It is also possible to introduce named predicates with either definitions or characterizations; this is discussed in 4.2.2.2. If the predicate "is_prime" has been defined, then the form

$$is\_prime(X2)$$

would assert that the value of X2 is prime. This is only meaningful if X2 is of integer type. These kinds of predicates, which assert a relationship among the objects of an object space, are used to specify initial states of machines and types, and preconditions of operations, programs, and actions.

Another form of predicate asserts a relationship between a state and its successor, that is, a property of a state transition. For example, the form

$$X1'=X1+1$$

characterizes the set of all transitions <q,q'> such that the value of element X1 in state q' is one more than the value of X1 in state q. That is, X1 has been incremented by one. The syntactic distinction is the occurrence of "X1"; the prime indicates value in the final state, and the unprimed occurrence indicates value in the initial state. This sort of predicate occurs in the specifications of operation, program, and action post conditions and event definitions. The complete specification of the square-root procedure sqrt(X:real, obj S:real) could be

pre X≥0
post S'*S'=X

The precondition states that the value X must be non-negative, and the post condition states that the final value of S is such that its square equals the initial value of X. Two important conventions are used here: (1) if a variable name X occurs only unprimed, then it is not altered it is as if the clause "X'=X" had been conjoined to the predicate, and (2) if a variable name does not occur in a predicate either primed or unprimed, then it is not altered. With this convention it is possible to check syntactically that the specification of a vfun (pure function) of a data type is indeed pure; the only name which may occur primed is that of the value returned.

Among the propositional connectives the grouping operators "[" and "]" have highest priorities, followed by the unary negation "not" ("~"). Next come "and" ("&"), "or," "xor," and last come "if ... then ..." ("=>") and "iff" ("<=>"). The parenthesized operators are alternate forms of the operators they follow in this list. The interpretation of these operators is the familiar one of propositional logic; briefly, the state q satisfies "not P(q)" if and only if q does not satisfy "P(q)".

- The state q satisfies "P(q) and R(q)" if and only if q satisfies both "P(q)" and "R(q)".
- The state q satisfies "P(q) or R(q)" if and only if q satisfies either "P(q)" or "R(q)".
- The state q satisfies "P(q) xor R(q)" if and only if q both satisfies "P(q) or R(q)" and does not satisfy "P(q) and R(q)".

- The state q satisfies "if P(q) then R(q)" if and only if either q does not satisfy "P(q)" or q satisfies "R(q)".

- The state q satisfies "P(q) iff R(q)" if and only if q satisfies both "P(q) ⇒ R(q)" and "R(q) ⇒ P(q)."

The terms which are related in a predicate may be object names, literal constants, or the cardinality operator "#". The term

#y1:T1, ..., yk:Tk[P(y1, ..., yk)]

denotes the cardinality or size of the set of all k-tuples <y1, ..., yk> of values which satisfy the predicate P. For example, the term

#i:integer[A(i)=0]

denotes the number of different integer values i such that the value of A(i) is zero; this is just the number of zero elements of A.

If n1, ..., nk are term names in a predicate then

(viii) forall n1:T1, ..., nk:Tk[P(n1, ...,nk)]

(ix) forsome n1:T1, ..., nk:Tk[P(n1, ..., nk)]

are respectively, the universal quantification of P and the existential qualtification of Pi "∀" is an alternate form for "forall" and "∃" is an alternate form for "forsome". Ni ranges over values of the associated type Ti. The interpretations are the conventional ones of first-order predicate calculus [ENDE72]:

- The state q satisfies "∀ni:T1, ..., nk:Tk[P(ni, ..., nk)]" if and only if q satisfies "P" for every valid assignment of values to ni, ...,nk.

- The state q satisfies "∃ ni:t1,...,nk:Tk]P(n1,...,nk]" if and only if q satisfies "P" at least one valid assignment of values to ni,...,nk.

For example, the predicate

∀i,j:integer[A.lob≤i<j≤A.hib=>A(i)≤A(j)]

asserts that, for every pair of index values i and j such that i<j, it is also true that A(i)≤A(j); that is, A is sorted in ascending order of indices.

When one of the ni's of (viii) or (ix) is preceded by the keyword obj, the interpretation is that ni ranges over the set of objects of type Ti in the machine containing the specification. Thus, if there were ten integer arrays A, ..., J, then the assertion that they were all sorted would be written

∀ obj Z:integer array, i,j:integer
[Z.lob≤i<j≤Z.hib=>Z(i)≤Z(j)]

An additional quantifier form is

for some unique ni:T1,...,nk:Tk[P(ni,nk)]

which asserts that there exists exactly one k-tuple of values for ni,...,nk which satisfy P. ∃! is an alternate form for this quantifier.

It is often necessary to introduce a temporary value in an assertion in order to relate it to other terms; this capability is supplied by the let clause. In the form

let pi:Ti:Tk such that P(pl,...,pk,)[S(pi,...Pk)]

it is being asserted (1) that there exist value pi of type Ti which satisfy "P" and (2) that every such set of values also satisfy "S". If this form occurs within some predicate, the predicates P and S may also use the terms of the containing predicate. Again, if Pi:Ti is preceded "obj" then the variable ranges over the set of object of type Ti in the machine containing the definition.

Behavioral Specifications -- An event is a set of state transitions, and an occurrence of an event is an element of that set. Given a predicate with primed

terms, the set of transitions which satisfy the predicate is the event characterized by it. Since a given event can occur many times during the history of a system, it is necessary to be able to indicate which occurrence of the event is being discussed. Let T be a predicate characterizing a state transition, for example the post condition at a program the form

effects (P(q,q'))<i>

denotes the i-th occurrence, in the history of the machine containing the specification, of a state transition satisfying P. For example, "effects(sort)<i>" denotes the i-th invocation of the program "sort". The integer expression between "<" and ">" is the event ordinal.

A syntactic variant is the "becoming true" of a predicate. The form

occurs (P(q))<i>

where P characterizes a state, is equivalent to effects(~P(q) and P(q'))<i>; it denotes the i-th transition from a state q which does not satisy P to state q' which does satisfy P.

The fundamental predicate relating events is precedes, which relates two events in time. The form

Event1<i> precedes Event2<j>

characterizes the set of all system histories in which, for some k and l, the i-th occurrence of Event1 is the k-th transition, the j-th occurrence of Event2 is the l-th transition, and k<l. All system histories in which either there is no order, or in which the i-th occurrence of Event1 follows the j-th occurrence of Event2, are excluded.

Two related predicates allow for guarantee of the future and guarantee of the past. The form

Event1<i> later Event2<j>

characterizes the set of all system histories with the property that either there is no i-th occurrence of Event1, or there is and it precedes the j-th occurrence of Event2. Similarly, the form

Event1<i> before Event2<j>

characterizes the set of all system histories with the property that either there is no j-th occurrence of Event2, or there is and the i-th occurrence of Event1 precedes it.

With propositional combination of these ordering relations, and quantification over the event ordinals, it is possible to specify constraints on the behavior over time of a system. For example, the assertion

```
Vi:integer [if i.0 then
  effects(read?)<i> later effects (read!)<i>
  and effects (read!)<i> before effects (read?)<i+1>
  ]
```

asserts that the transitions "read?" and "read!" occur in the order read?...read!...read?...read!...; this could formalize the requirements that "read" transactions to a data base be mutually exclusive.

There are primitive predicates associated with inlet and outlet data types; they are "gets" and "arrives" for inlets, and "puts" and "leaves" for outlets. These allow for the specification of communication properties among machines; for example, the specifications

(x)
```
Vi:integer [if i≥ 0 then
effects (leaves(c.out))<i> later (arrives(c.in))<i>
]
```

asserts that the i-th departure from the outlet of channel c is always followed by the i-th arrival at the inlet end of c. This is not enough by itself to specify reliable, order-preserving flow of information along the channel; it is necessary to be able to talk about the values of objects and expressions when particular events occur. The form

Value prior Event1

denotes Value at the initial state of Event1; similarly, the form

Value after Event1

denotes Value at the terminal state of Event1. Thus, the value which leaves an outlet "out" is the contents of the window at the start of the i-th leaves event; this would be expressed by the term

out.window prior effects (leaves(out))<i>

and the reliability of communications for (x) above would be expressed by

```
Vi:integer[if i> 0 then
        c.out.window prior effects (leaves(c.out))<i>
        =c.in.window after effects (arrives(c.in))<i>
        ]
```

4.2.2.2 Declarations -- It is convenient to be able to introduce named expressions, possibly with parameters, for the various kinds of specification forms. In this section the mechanisms for introducing such named forms are defined.

Predicates -- The form

PredicateName (Params) means PredicateExpression

declares a predicate with formal parameters to mean the expression on the right. This expression may be discussed so far which characterizes a state, including terms related to events with prior or after. Alternatively, the right hand expression may be the keyword "abstract." This allows for the introduction of predicates which are used at one level but defined at a lower level. To allow for predicates whose definitions would be unwieldy, there is also provision for declaring a set of properties (axioms) which characterize the predicate. In the form

PredicateName (Params) characterized by PredicateExpression

the PredicateExpression is typically a set of predicates joined by "and"; they would be used in a demonstration of correctness or consistency.

Actions -- The form

PredicateName (Params) means pre Assertion post Assertion

allows a predicate characterizing a set of state transistions, such as a program precondtion-postcondition pair, to be defined. The rules for the two assertions are the same as those for program specifications.

Flows -- The form

FlowName:Type from Object1 to Object2

declares FlowName to be the name of a logical flow of information of type Type from Object1 to Object2. Object1 and Object2 must be the names of objects in the machine containing the declaration. This allows a textual specification of data dependencies among members of a set of data objects before that set is partitioned.

4.2.2.3 Mappings -- In 3.4.2 three kinds of hierarchical organizations corresponding to data, programs, and machines were discussed. The CSDL descriptions of a system uses all three of these structures. Since a step along the hierarchy indicates a refinement step in the design process, there is in general a mapping problem, which is to show that the abstract and refined views of an object, program, or machine are consistent. Thus, the specification language must support the specification of those mappings.

Data -- The form

let GenericName:Type;
Representation

defines a mapping function from the lower-level representing machine onto the conceptual object space of the data type to be represented; hence, the mapping

function is defined within the representing machine. The Representation part is a semicolon-separated list of clauses of one of the two forms

Lower represents Upper, or
if Condition then Lower represents Upper

The Condition may be either a predicate over the representing object space, or a "Lower represents Upper" phrase. The Condition allows the specification of mappings dependent on the state of the representing data structure, for example the relative position of pointers in a circular buffer representation of a bounded queue, as well as structural dependencies, such as the relation between pointer chains in a linked list and successive indices of a file viewed as an array.

The Upper portion may be either a predicate name with formal parameters, or a component of the conceptual object space (or an attribute of such a component). In the first case, the Lower portion is a predicate over the representing object space; the intent is to define the abstract predicate declared at the upper level in terms of the representation. For example, the predicate "precedes (x,y: record)" could be characterized as a partial order relation at the upper level even though the type "record" is abstract; the predicate would be defined in the machine which refines the file and record types. If the Upper portion of a representation clause is a component of the conceptual object space then the Lower portion is a value expression expressed in terms of the representing data structure.

The mapping function may be used by uniformly substituting Lower portions for Upper portions which occur in the specifications of the type. The resulting assertions constitute specifications of initial states, data invariants, and program specifications which must be satisfied by the representing data structure and the programs which implement the type operations in order for the refinement to be a consistent representation of the data abstraction.

Programs -- In the CSDL organization of a simple machine, the description of a given procedure does not appear in a sequence of more refined forms; rather, it appears once as text in the algorithmic design language. Hence, there are no mapping functions associated with the description of a program.

Machines -- After the data of a machine have been partitioned among component machines, and the interface objects connecting them have been described, the specifications of behavior defined over the unpartitioned data must be mapped onto specifications of the components and interfaces. Specifications of initial states and data invariants become specifications of the components which contain the associated data objects. Behavioral specifications concerning data objects which are placed in the same component likewise become specifications of the component machines. All of these specifications are placed in the descriptions of the component machines. There remain the specifications which refer to data which are placed in different machines; these become specifications of the interactions among the component machines.

The form

FlowName passes through InterfaceNames

specifies that the logical flow of information named "FlowName" is to be accomplished by going through the ordered list of interface elements listed in InterfaceNames. This allows the factoring of a flow between objects in different components into (1) flows within each component between the "end" objects and elements of interfaces within each component and (2) flows within the interface objects.

The form

function Assertion

is used to specify the intermachine, or communication, behavior. This specification is contained in the partitioned machine. It expresses the behavior of the interface objects which connect its components. Here is expressed the requirements on the communication interface which must be statisfied in order for the several components to interact, each satisfying its own requirements, so that the required behavior of the entire partitioned machine is obtained.

4.3 ORGANIZATION OF A CSDL DOCUMENT

4.3.1 Principles of Organization

A system in CSDL is made up of a collection of machines. These machines form a hierarchy based on the principles of object space decomposition and of type refinement.

4.3.1.1 Object Space Decomposition -- The object space of a machine, consisting of a collection of data objects together with the logical paths of information flow among the objects, may be partitioned into disjoint components. For a machine whose object space is not subdivided, the document includes the definition of a hierarchy of programs that realize the specified behavior of the machine.

If the state space is partitioned, every object is placed into a unique subspace. Wherever a logical path of information flow crosses the component boundaries, an interface object is introduced to provide the necessary interface among distinct components. The outlet end of a channel is placed in the component containing the source of an information flow, and its inlet end is placed in the component containing the destination of the information flow. After a partition has been described, a machine is defined over each component space, and these component spaces may themselves be subdivided.

To illustrate, suppose A and B are two objects with a logical path lp of information flow between them. Figure 11 shows the decomposition of this object space into two components, and the introduction of the channel object C1 at the boundary. The design decision here is that two machines will be defined: one over A and the outlet end of C1, and the other over B and the inlet end of C1. Figure 12 shows the decomposition of the same object space into three components, with channels C1 and C2 introduced at the boundaries. The design decision in this case is that an additional machine will be defined over the middle component, consisting of the inlet end of C1 and the outlet end of C2, whose sole purpose is to manage the passage of information between the other two machines.

In CSDL, all the resources making up a system are treated by the concept of data objects. This practice permits the partitioning activity to take place prior to the allocation of hardware, firmware, and software resources.

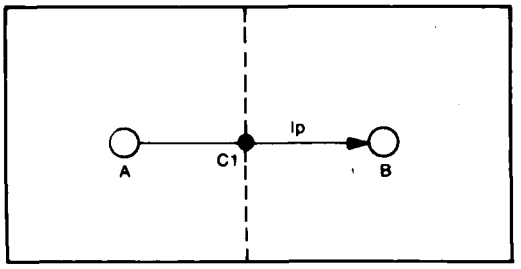


Figure 11. Decomposition into Two Components

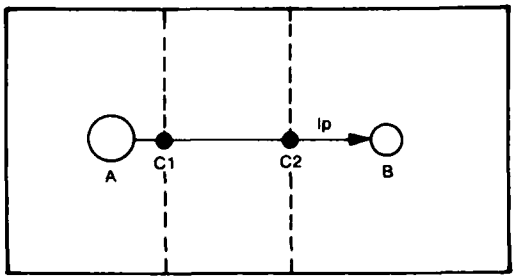


Figure 12. Decomposition into Three Components

4.3.1.2 Type Refinement -- The use of type refinement in CSDL permits the incremental design of a system. First, machines (either partitioned or unpartitioned) are defined over objects of some high-level type. Then, the data type is refined by defining a machine whose object space replaces a generic object of the refined type. All the operations on the refined type are represented by either programs or type operations at the lower level, and the type specifications at the upper level are re-expressed in terms of those at the lower level. After the refining machine has been specified, its object space may also be partitioned, if so desired.

Two kinds of type refinement are identified: data expansion and data representation. The fundamental idea of data expansion is to invent an abstract data type and then to refine it by data and programs at a lower level. An example, shown in Figure 13, may be a data type called "archive" defined as an array of objects of the type "file", which in turn is abstract. In the machine where this type is defined programs will be written using functions of the type "file." The type "file" is then refined in a lower-level machine where "file" is described as an array of "record". The functions of the type "file" are now refined in programs which ignore the existence of the type "file", but are operating on objects of the type "record".

In data representation, a data structure used to define a data type is mapped onto another data structure, usually because the latter is closer to the one available in implementation, while the former permits an easier definition of the problem. In the example shown in Figure 14, a file at the upper level of design is represented as a sequence of objects whose type is "text". This file is represented at a lower level as a

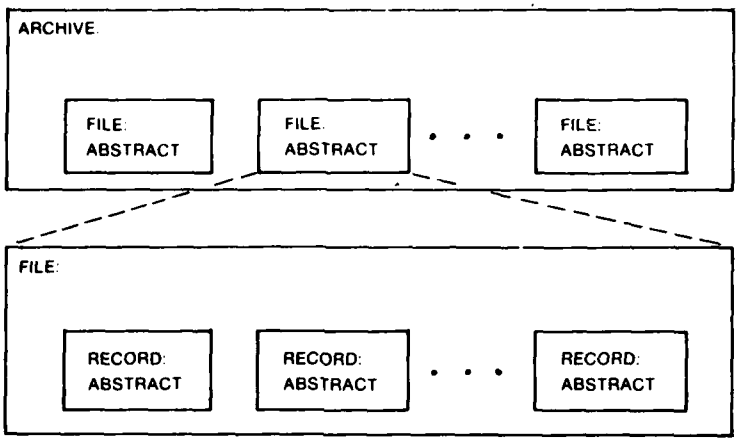


Figure 13. Example of Data Expansion

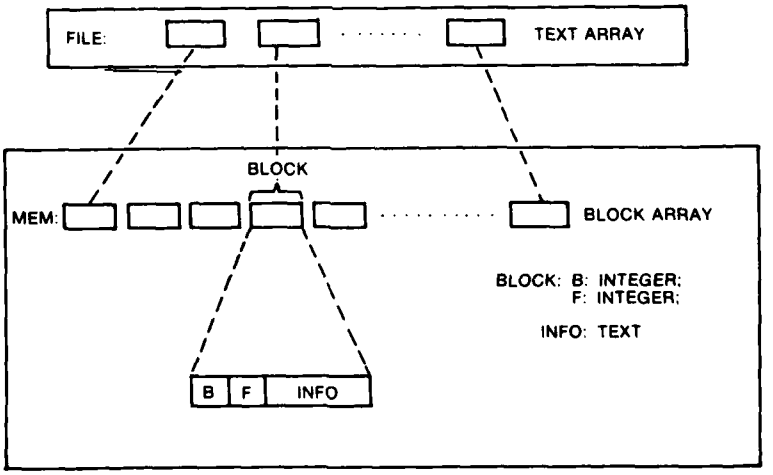


Figure 14. Example of Data Representation

linked list of blocks, where the type "text" is supplemented with forward and backward pointers. Thus, at the upper level, the fact that the file was to be represented as a linked list was transparent and perhaps unknown. Note that it is not necessary for all of the blocks in the linked list to correspond to elements of the text array.

The data types used to refine a higher-level type may themselves be refined by types of some yet lower level. The type refinement activity extends down to machines whose data types are considered primitive either because the hardware (or software) supplies those data types or because their representation is standardized in some form.

4.3.1.3 An Illustration -- Designing a system consists of interleaving the activities of object space decomposition and type refinement. This creates a hierarchy of machines (an example of which is shown in Figure 15), and a design is complete if at the bottom of the hierarchy, programs are designed for every machine, and each machine has in its object space only objects of primitive types.

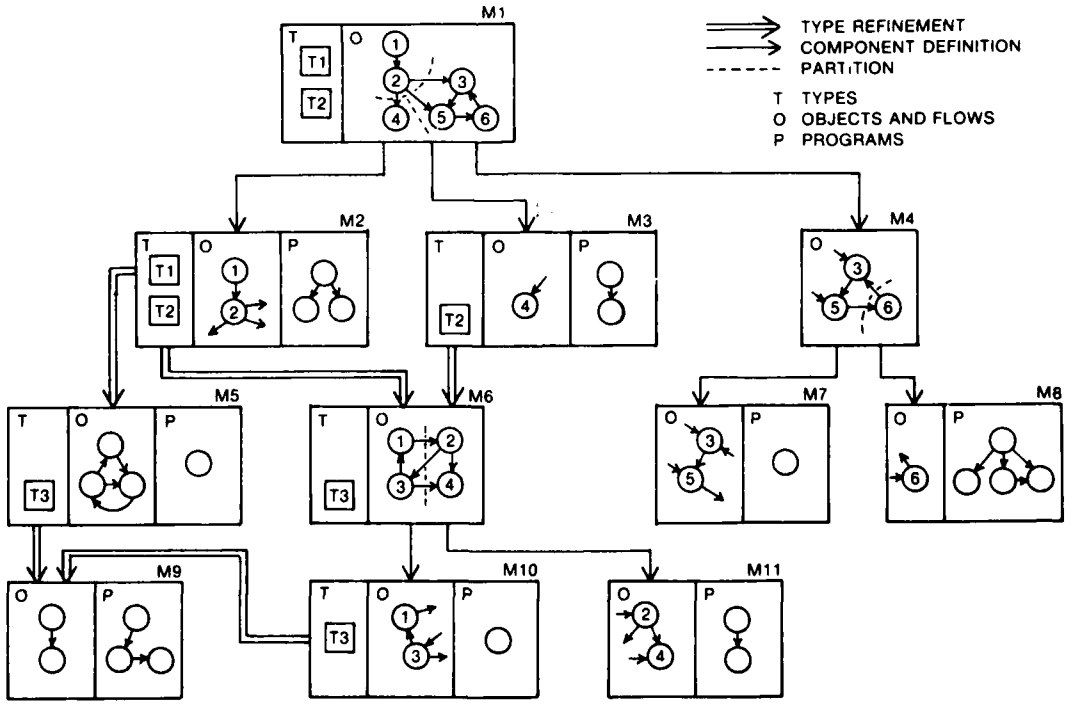


Figure 15. Hierarchy of Machines in CSDL

In Figure 15 each machine shown consists of up to three parts: data types, data objects with paths of information flow among them, and programs. The object space of machine M1 is partitioned into three components (this partitioning is indicated in the figure by the dotted lines), and machines M2, M3, and M4 are specified over the three components. Likewise, the object space of M4 is subdivided, and machines M7 and M8 are specified over the components. These machines constitute a hierarchy for component definition. Figure 15 also shows the hierarchy for type refinement. Types T1 and T2 are refined by machines M5 and M6, respectively, in terms of objects of Type T3 and some primitive types. The object space of M6 is subdivided into two objects spaces for machines M10 and M11. Finally, type T3 is refined in terms of the objects in machine M9, and at this point the design has been completed.

4.3.2 Elements of a System Definition Text

The definition text of a system, as illustrated in Figure 16, is enclosed by the keywords system <system id> and end <system id>, where <system id> is the name of the system being documented. It contains primarily a collection of machine definition texts arranged in some orderly fashion. Also included is a list of identifiers of all the machines making up the system. Each machine identifier may, if so desired, be followed by a description, enclosed in curly braces and in plain English, of the corresponding machine.

4.3.3 Elements of a Machine Definition Text

A machine definition text consists of the declaration of a state space of data objects together with the specification and description of the static and dynamic behaviors of the machine over its object space. This document is organized into sections enclosed by the keywords machine <machine id> and end <machine id>, where <machine id> is the name of the machine being documented. Figure 17 shows the structure of documentation for a generic machine.

It should be pointed out that the purpose of this subsection is not to advocate a format for machine documentation but rather to describe a machine document that includes, in an organized fashion, all the technical information related to the design of a machine. The particular style of documentation format being presented has the

```
system <system id>

     <machine id>
     <machine id>
     .
     .
     .

<machine definition text>
<machine definition text>
 .
 .
 .

end <system id>
```

Figure 16. Format of System Definition Text

```
machine <machine id>
     refines <type id>

declarations
     predicates
     types
     objects
     actions
     flows
end declarations

specifications
     mappings
     states
     behavior
          function
end specifications

partition
     interfaces
     paths
     components
     communication behavior
          function
end partition

programs
     <program definition text>
     <program definition text>
     .
     .
     .
end programs

end <machine id>
```

Figure 17. Format of Machine Definition Text

property that every aspect of the machine is first declared, then specified, and finally described. The location of performance specifications in relation to this documentation format is discussed in 6.3.5.

The declarations and specifications are grouped into the declarations and specifications chapters, respectively. The descriptions are contained in either the partitions or programs chapter. The partition chapter is found whenever the object space of a machine is partitioned, and the programs chapter exists only if the machine being described is not partitioned.

Comments or informal explanatory notes may appear anywhere within the body of the machine definition text. These are differentiated from the formal text by enclosing them in curly braces.

The keyword refines is used to indicate the machine as one whose object space is the refinement of a higher-level type; <type id> gives the name of the type being refined. For closely related nonprimitive types, it may be convenient to include the refinement definition of them within the documentation structure of one machine. This is done by prefixing the refinement of each type by a new refines <type id> heading in the manner shown in Figure 18. In this case, the semantics is that wherever an object of one of the refined types occurs, it is refined by the appropriate object space in this machine.

Without the keyword refines the machine being defined is one whose object space is the detailed definition of a component space of another machine, or one whose object space is the global space of an entire system.

4.3.3.1 Declarations Chapter -- The predicates section contains the declaration of predicates which are convenient to define once and then use in the specifications. For each predicate, a name is associated to an assertion that is expressed in terms of the CSDL specification language. For example, a predicate SORTED can be defined here in detail for an array and subsequently used by writing only the word SORTED followed by the appropriate parameters.

```
machine <machine id>
     refines <type id>

<declarations chapter>
<specifications chapter>
 .
 .
 .

     refines <type id>

<declarations chapter>
<specifications chapter>
 .
 .
 .

end <machine id>
```

Figure 18. Format for Refining Multiple Types

The types section contains the definition of the nonprimitive types needed in this machine. These types may be refined by lower level machines. The definition of a type consists of the specifications for the permissible functions on any object of the type and for all of the invariants that must be preserved on those objects.

The objects section declares the object space to be used in this machine. The declaration of a object consists of giving it a name and specifying its associated type, which may be the name of a defined type in the types section.

The actions section contains the declaration for actions that can be used in the specifications chapter. For each action, a name is associated with a pair of assertions expressed in terms of the CSDL specification language. Constraints on the initial states of an action are characterized under the pre heading, and the desired relations between its final state and the initial state are characterized under the post heading.

The flows section contains the declaration of the logical paths of information flow among the objects declared in the objects section.

4.3.3.2 Specifications Chapter -- The mappings section exists only for machines whose object space is the refinement of higher-level types. Here the relationships between the objects in this machine and those of the types being refined are established. If

predicates on objects of the refined type were defined, their meaning must be restated in terms of the objects in this machine. All operations on the refined type must also be re-expressed in terms of the programs or the type operations in this machine. The basic building block of this section is the representation clause explained in 4.2.2.3.

The states section contains, under the init heading, static assertions which characterizes the initial state of the machine. This includes both the initial values of the data objects and the initial relationships among them. Under the final heading are static assertions that specify the desired relationships between the final and the initial states. Under the invariant heading are conditions which must be preserved throughout the existence of the machine. The assertions are expressed in terms of the CSDL specification language.

The behavior section includes, under the function heading, the temporal assertions on event sequences which must be maintained throughout the existence of this machine. It also specifies the relationships among data values at specific instances of time when events occur. The specifications are expressed in terms of the CSDL specification language.

4.3.3.3 Partition Chapter -- The interfaces section introduces objects, such as channels, needed to provide the necessary interface among the components making up a partition. A name and its associated type are given for each interface object declared here.

The paths section specifies a set of paths that establishes the location of the interface objects relative to the other objects. The data type of an interface object must, of course, be identical to the type of information that passes through it.

The components section contains the declaration of the components making up the partition. Each declaration gives some component a name, lists the objects that are found in the component, and gives the name of a machine defined over the component. The objects, then, comprise the entire object space of that machine. For each object in the list there is a corresponding object (of the same type) declared in the objects section of that machine.

The communication behavior section contains, under the function heading, the static and temporal specifications which are not derivable from those in the states and behavior sections, but must be included as a consequence of the object space decomposition. It may also include the restatement of the static and temporal relationships among objects in distinct components in terms of their relationships to the newly declared interface objects. These specifications, once again, are in terms of the CSDL specification language.

4.3.3.4 Programs Chapter -- This chapter contains the definition of a hierarchy of programs which, as a group, realize the specified behavior of the machine. The program at the top of the hierarchy is, by convention, named "controller" and placed first in the chapter. All the other programs, if any, can be arranged in some orderly fashion.

4.3.4 Elements of a Program Definition Text

The format of a generic program definition text is shown in Figure 19 where <program id> is the name of the program being described.

```
program <program id> (<input parameters>)
      returns <output parameter>
pre
post
invariant
variables
text
end <program id>
```

Figure 19. Format of Program Definition Text

The pre section specifies the permissible machine states when invoking this program.

The post section specifies the state of the machine when the program terminates properly. Also included are the relationships between this final state and the pre state.

The invariants section specifies the conditions on the objects in the machine that must be preserved by this program. These specifications are expressed in terms of the CSDL specification language.

The variables section contains the declaration of all the local data of the program. These data, which may be of a type declared in the types section, exist only for the duration of this program.

The text section contains the description of the program using the algorithmic language presented in 4.2.1.3.

SECTION 5
AN ILLUSTRATION OF SYSTEM DEFINITION IN CSDL

In this section, system definition techniques for CSDL are illustrated by stepping through an example of a concurrent system design. In the description of the example, formality and completeness are sacrificed for the sake of comprehensibility. The example used to demonstrate the system definition techniques is the Towers of Hanoi game.

The Towers of Hanoi is a game in which three spindles are used to host a set of n rings which are all different in size. Each spindle can host all the rings or a subset of the rings. As shown in Figure 20, the object of the game is to start with all rings stacked on one spindle and to build up the same configuration on another spindle. The third spindle is used to make the move possible. The spindles are called ORIGIN, INTERMEDIATE, and DESTINATION to indicate their intended use.

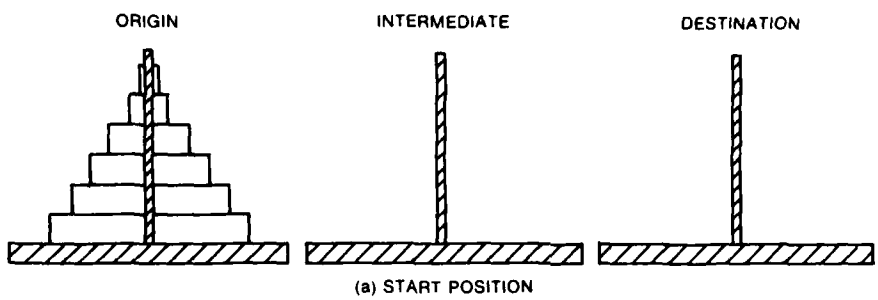
The game rules put the following constraints on the possible moves:

- o No larger ring can ever be placed on top of a smaller ring.
- o Exactly one ring can be moved at a time.

The moves required for a Towers of Hanoi game with four rings, i.e., n=4, are shown in Figure 21. The rings are represented by integer numbers indicating their size. The configurations shown are preconditions for the moves indicated by dashed arrows.

The Towers of Hanoi game is a trivial example for which a simple sequential solution exists in the form of a recursive algorithm. For the illustration of a system definition in CSDL, a partitioned and concurrent solution will be derived.

Although the structure of the game suggests a partitioning into three separate towers, it will be demonstrated how CSDL supports the derivation of such a partition. Thereby, the example will demonstrate how disjoint state spaces and the communication among the machines associated with these state spaces are handled in



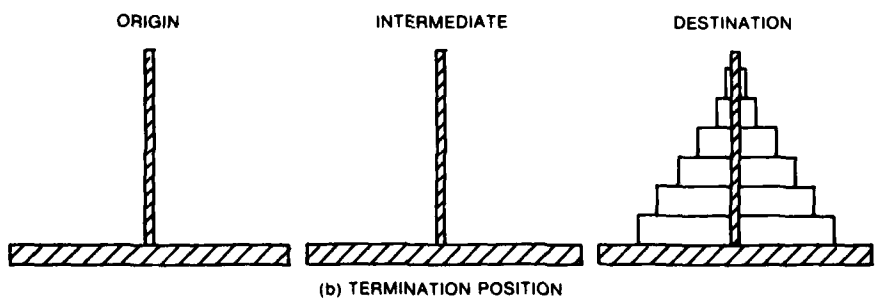


Figure 20. Towers of Hanoi

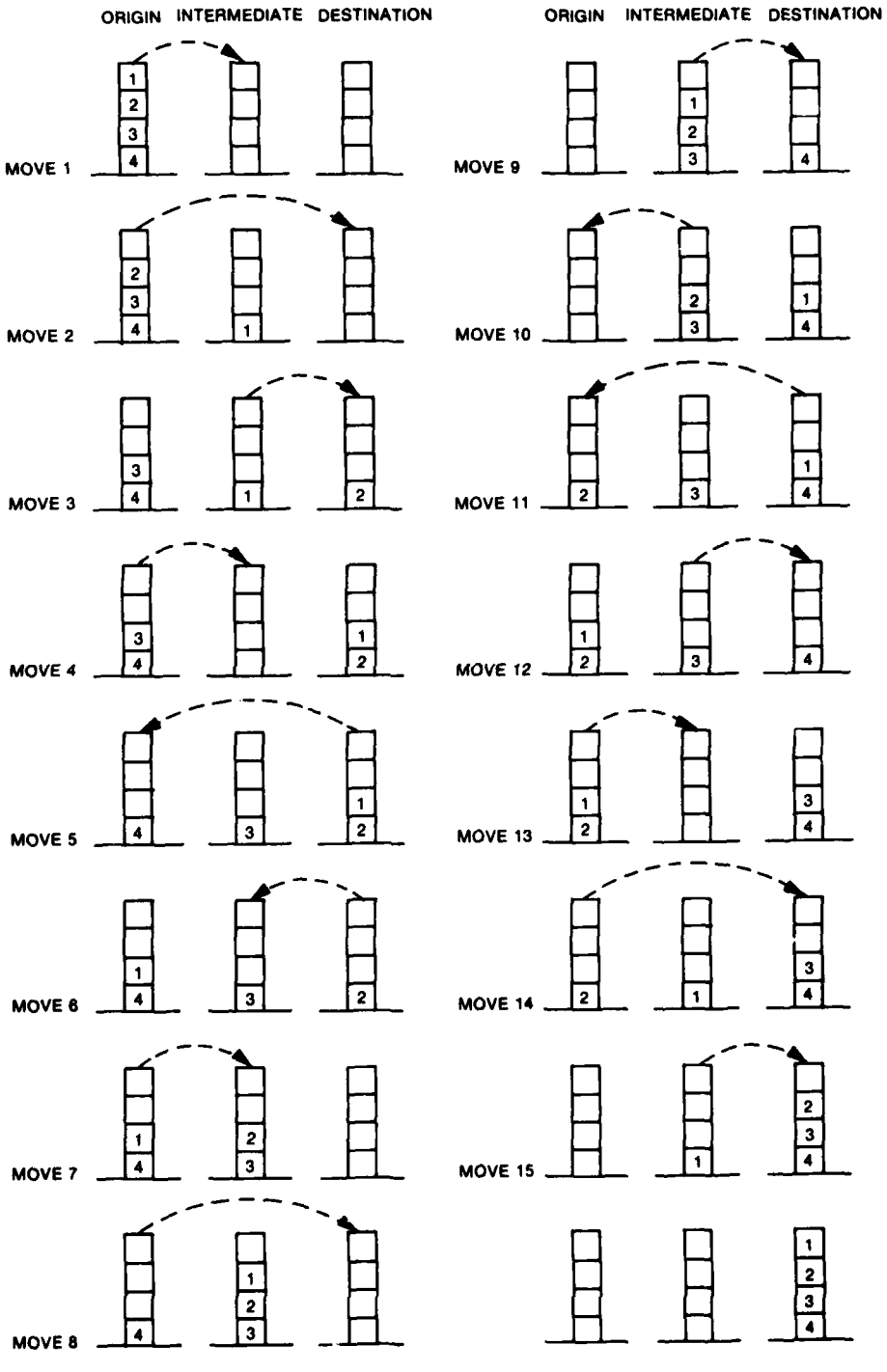


Figure 21. Solution of Towers of Hanoi for Four Disks

CSDL. It will also be demonstrated how to incrementally present a system definition as a hierarchy of CSDL machines and how to introduce a communication structure for a partitioned and concurrent system design.

Due to the simplicity of the example, only a flavor of the application of CSDL to real world systems can be conveyed here. An illustration of the use of CSDL for the definition of more complex systems is included in Appendix D.

5.1 TOP LEVEL DESIGN

This subsection describes a system definition for the Towers of Hanoi game. The definition of this system is bracketed as shown in Figure 22.

The top-level definition of a system is a machine which specifies the requirements on the system design and describes the associated system design. The system design may be given in the form of a simple or partitioned machine. For the example in this chapter, we presume a partitioned solution which is documented as shown in Figure 23.

The machine "three_towers" is parameterized by the integer n, which is the number of rings in the game. That is, the definition of the machine, "three-towers", can be instantiated for a particular value of n.

5.1.1 Declarations

The structure of the CSDL declarations chapter used in the Towers of Hanoi example is shown in Figure 24.

The first step in deriving a solution to the Towers of Hanoi game is to review the game rules. To express these rules, data objects need to be defined which represent the three spindles and the rings to be moved. To this end, abstract data types for spindles and rings need to be defined. Therefore, the first step in the description is the definition of the data types "ring_size" and "tower" given in Figure 25. Additionally, an abstract data type "posint" is defined.

```
system towers_of_hanoi

     {definition of the system "towers_of_hanoi"
     including all levels of partitioning and
     refinement, i.e., this system definition includes
     all partitioned and simple machines needed for the
     description}

end towers_of_hanoi
```

Figure 22. Outer Brackets of a System Definition

```
machine three_towers(n:integer)
     declarations
          {declaration of predicates, types, objects and
          actions needed to define the machine}
     end declarations
     specifications
          {specification of the functional and temporal
          behavior the machine posses}
     end specifications
     partition
          {definition of the partitions of the machine and
          their interfaces and interrelations}
     end partition
end three_towers
```

Figure 23. Outline of the Top-Level Machine for a Partitioned Solution of Towers of Hanoi

```
declarations
     predicates {definition of statements to be used in the
     system description}
     types {definition of the data types
     of the data objects of the machine}
     objects
     {declaration of the data objects constituting the state
     space of the machine}
     actions {specification of
     activities to be used to define the machine behavior}
end declarations
```

Figure 24. Structure of the Declarations Chapter

```
types
     posint:integer
          let pi:posint
          invariant
               pi>0
     end posint;

     ring_size:integer
          {the rings are represented by their size which is given
          by an integer number between 1 and n; note that n is a
          parameter of the machine being defined}
          let rs:ring_size
          invariant 1 ≤ rs ≤ n
     end ring_size;

     tower: ring_size array
          {a spindle is represented by an array of rings}
          let t:tower
          invariant
               {the rings on a spindle are ordered by their size
               such that no larger ring is on top of a smaller
               ring}
               forall i,j:posint [if INDEX(t,i) and INDEX(t,j)
                    then [if i<j then t(i)>t(j)]]
               and t.dom≤n
     end tower;
```

Figure 25. Type Declaration

The predicate INDEX used in the declaration of type "tower" is to be defined in the predicates section. Its declaration is shown in Figure 26.

Given the type declarations of Figure 25 the state space of the machine, three_towers, can now be defined by the object declarations shown in Figure 27. In correspondence with Figure 20, "org" is the spindle containing all the rings at the start of the game, "dest" contains all rings at the end of the game, and "int" is the intermediate spindle.

With the definition of a global state space in place, the next step in the design is the identification of actions which need to be carried out. In the case of the Towers of Hanoi game, two basic actions are suggested by intuition. The first action is the construction of (sub)towers on either one of the three spindles. The second action is the movement of a single ring from one spindle to another. The declaration of these two actions is given in Figure 28. Deviating slightly from the formal syntax definition, English text is used to describe the input and output assertions.

Given the declaration of predicates, types, objects, and actions developed in this section, the next step in the description is the specification of the behavior of the machine "three_towers".

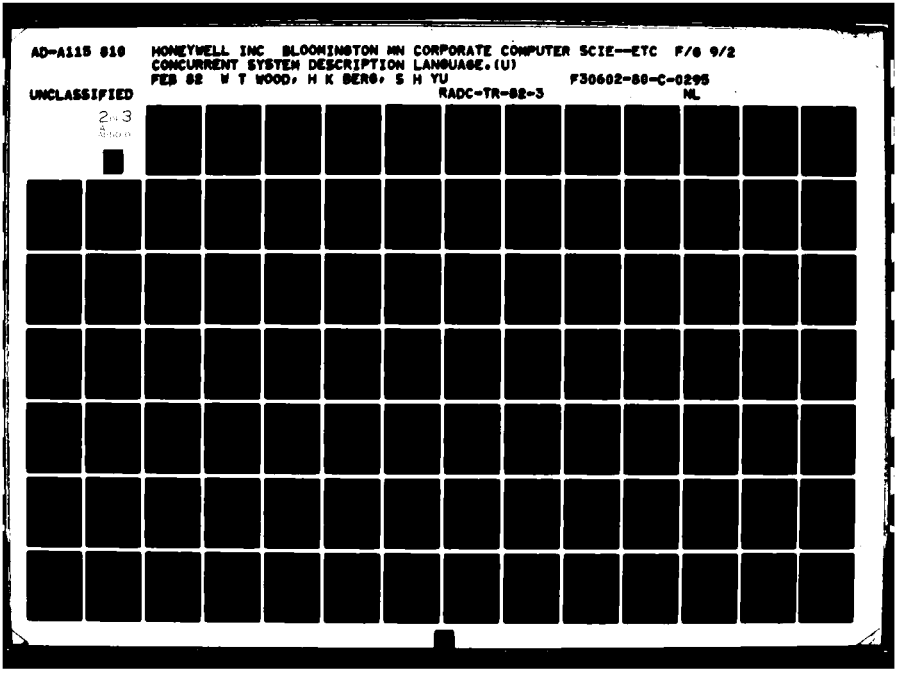
5.1.2 Specifications

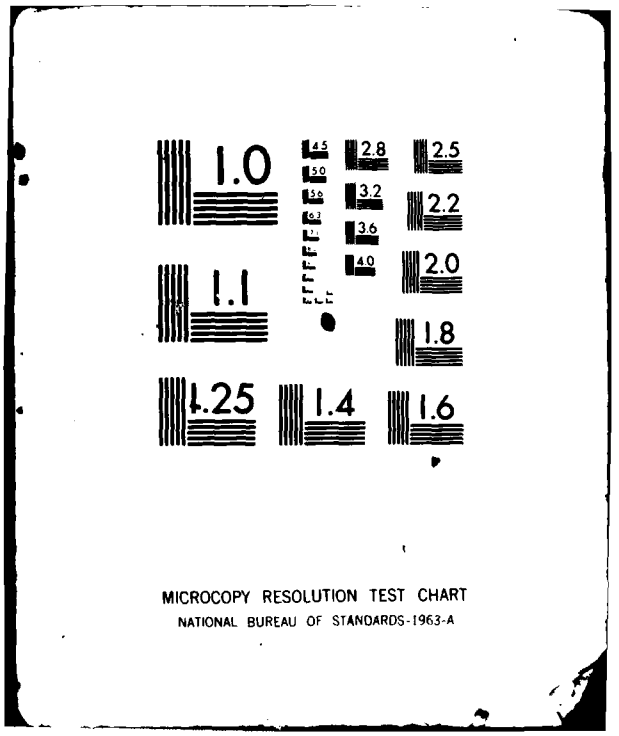
The structure of the CSDL specification chapter used in the Towers of Hanoi example is shown in Figure 29.

The Towers of Hanoi game represents a terminating computation. Thus, the states specification consists of an initial and final assertion. These assertions can directly be derived from the game rules. The states specification is shown in Figure 30.

The next step is the specification of the behavior. Here, the game rules and requirements for computational progress are to be defined in terms of temporal relationships between instantiations of the declared actions. To guarantee the game rules, the following properties need to be preserved.

AD-A115 010 HONEYWELL INC BLOOMINGTON MN CORPORATE COMPUTER SCIE--ETC F/G 9/2
CONCURRENT SYSTEM DESCRIPTION LANGUAGE.(U)
FEB 82 W T WOOD, H K BERG, S H YU F30602-80-C-0295
UNCLASSIFIED RADC-TR-82-3 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
predicates
     INDEX (A:typename array, i:posint)
          means A.lob ≤ i ≤ A.hib;
```

Figure 26. Predicate Declaration

```
objects
     org:tower;
     int:tower;
     dest:tower;
```

Figure 27. Object Declarations

```
actions
     construct (large:ring_size, small:ring_size, spindle:tower)means
               {construction of a subtower on "spindle" with ring
               "large" as the base and ring "small" as the top}
     pre ["spindle" is empty] or [the top of "spindle" is larger
          than the base of the subtower to be built]
     post [the subtower has been constructed on top of
     "spindle"];

     move (ring:ring_size, spindle1:tower, spindle2:tower)means
               {move a single ring from "spindle1" to "spindle2"}
     pre ["ring" is on top of "spindle1"] and
         [the ring on top of "spindle2" is larger than "ring"]
     post ["ring" is removed from "spindle1" and placed on top of
          "spindle2"];
```

Figure 28. Action Declaration

```
specifications
     states
          {specification of the initial state, final state,
          and invariant conditions}
     behavior
          {specification of temporal ordering constraints}
end specifications
```

Figure 29. Structure of the Specifications Chapter

```
states
     {specification of the initial state of all three spindles}
     init org.dom=n and int.dom=0 and dest.dom=0 and
          forall i:posint [if INDEX(org,i)
                    then org(i) = org(i-1)-1];

     {specification of the state reached after the termination of
     the last action} final construct (n,1,dest);
```

Figure 30. States Specification

- o Any move involves a single ring only, i.e., subtowers containing more than one ring cannot be moved.
- o Any ring to be moved must be on top of the tower from which it is moved.
- o No larger ring can be placed on top of a smaller ring.
- o When a ring is moved, all smaller rings must be on the spindle not involved in the move.

The properties listed above are ensured by the definition of the action "move" and the type "tower".

To guarantee computational progress, the following properties need to be preserved.

(1) At any given time, exactly one ring is being moved.

(2) No move must reverse the preceding move.

(3) Except for the construction of the full tower on "dest", the construction of a subtower with a base of size i must be followed by the move of the ring with size i+1 between the other two towers.

Property (1) can be expressed by stating that any move must be concluded before the next move starts. Property (2) requires that following a move of a ring, a different ring must be moved, before the first move can be reversed. Property (3) can be expressed by stating that the construction of a subtower with base of size i and top of size 1, must be followed by the move of the ring with size i+1 to the spindle on which the construction of a subtower with a base larger than i is in progress. The formal specification of these properties is given in Figure 31.

5.1.3 Partition

The structure of the CSDL partition chapter used in the Towers of Hanoi example is shown in Figure 32. The following procedure is used to derive the partition:

- o Identification of objects to be associated with the components
- o Definition of the communication interfaces
- o Definition of the components with their communication interfaces
- o Specification of the communication behavior

```
behavior
function

let t1,t2,t3,t4:obj tower

[{at any time, only one ring is being moved}
forall k,l:posint[
     forall i,j:ring_size[if i≠j
          then [if first move(i,t1,t2)<k>precedes first move(j,t3,t4)<l>
               then last move(i,t1,t2)<k>precedes last move(j,t3,t4)<l>]
]]
and
{no move must reverse the preceding move}
forall i,j:ring_size[if i≠j
          then move(i,t1,t2) later move(j,t3,t4) before move(i,t2,t1)]
and
{except for the construction of the full tower, any construction of a
subtower is nested in the construction of a larger subtower}
if t1≠t2 and t2≠t3 and t1≠t3
then forall i,j:ring_size[if 1<i≤n-1 and 1<j<2
               then last construct (i,1,t1) later move (i+1,t2,t3)
                    before last construct (i+j,1,t3)]];
```

Figure 31. Behavior Specification

```
partition
     interfaces
          {declaration of channel objects for communication among
          the components}
     components
          {definition of the components in the partition}
     communication behavior
          {specification of rules for the communication among
          components}
end partition
```

Figure 32. Structure of the Partition Chapter

The action declarations (see. Figure 28) suggest a partition into three components. Each component contains one of the three spindles. Assuming such a partition, the action "move" needs to be mapped into communication among two components. Furthermore, the computations performed in each component can be based only on the local state, i.e., the contained spindle, and the "messages" received from the other two components.

A possible scenario for the Towers of Hanoi game with three independent towers is the following. For the construction of subtowers, each component can sell and buy rings. When a ring is bought, it is placed on top of the local tower. When a ring is sold it is removed from the top of the local tower and sent to the component which bought the ring. That is, the action "move" is transformed into a deal between two towers. Several precautions must be taken to guarantee an orderly conduct of the game. For example, the desire of a component to sell a ring must be made known to the other two components, as they know only the state of their local spindle and the mailboxes through which they receive messages. If a sale is announced, both components receiving the announcement may desire to buy the appropriate ring. For the selling component to decide to which component to sell the ring, it is required that all buyers send a bid for the deal. After a decision has been made, a deal is closed with one of the bidders and a negative confirmation is sent to the other bidder. Each component decides on the basis of its local state as to whether it wants to issue a bid for a ring on sale. The component containing the spindle "dest" issues a "stop" command to terminate the game.

Figure 33 contains a declaration of data types suitable for the definition of a communication structure for the scenario described above. These data type declarations are added to the types section shown in Figure 25. The data type "com_struc" defines the entire interconnection structure for the communication among components. Therefore, the interface declaration in Figure 34 contains only a single element. A refinement of the communication structure into a communication subsystem is described in 5.3.

Having identified the data to be associated with the components in the partition and given the declarations of the data types for the communication structure, the

```
dealing:(sale[]bid[]deal[]nodeal[]stop);
        {message component used to synchronize deals between
        components of the concurrent system}

message:(op:dealing,ring:ring_size);
        {message structure used for communication among
        components}

com_struc:message channel array
   let cs:com_struc
   invariant cs.lob=1 and cs.hib=6 and
      forall i,j:posint [
         if INDEX(cs,i) and j>0
         then effects (leaves(cs(i).out))<j>
            before effects (puts(cs(i).out))<j+1>
      ]
   {the channels are sufficiently fast that it is
    unnecessary to check for departure}
end com_struc;
```

Figure 33. Data Types for Communication Structure

```
interfaces

      net:com_struc;
```

Figure 34. Interface Declaration

components and their communication interfaces can now be defined. The component declarations are given in Figure 35 The resulting system structure is shown in Figure 36.

The next step in the design is to specify the behavior which is relevant to the information flow in the communication network. Analysis of Figures 30 and 31 reveals that the following requirement on the communication behavior needs to be expressed.

At any time, exactly one ring can be moved. This requirement maps into the communication structure as follows. For any two components which close a deal (i.e., send a message of the form "deal,i" to another component), the deal of one of the components must be closed, before the "deal" message of the second component is sent out. The specification of this requirement is given in Figure 37. The predicate DEALS (m:message outlet) used in Figure 37 is defined to mean "puts (m) and m.window.op=deal".

Due to the partitioning and the introduction of the communication structure, an additional requirement needs to be specified. This requirement states that whenever the stop command is issued, it is issued by "tower 3". This requirement is also stated in Figure 37.

5.2 DESIGN OF MACHINES IN THE PARTITION

This subsection illustrates the design of the three machines which define the three components, "tower 1", "tower 2", and "tower 3", declared in Figure 35. The resulting system structure of the CSDL definition is shown in Figure 38. Obviously, these three machines are very similar. Therefore, the entire definition is presented for only one of the three machines, and necessary modifications for the other machines are discussed. The full definition is given for the machine "tower_org".

5.2.1 Declarations

The machine "tower_org" is defined in the outer bracket shown in Figure 39. The types used in the machine "tower_org" are the same as those declared in the top level

```
components

        tower1:(org,
                net(1).out,
                net(3).out,
                net(4).in,
                net(2).in):tower_org;
        {component containing spindle "org" of the global state space}

        tower2:(int,
                net(4).out,
                net(6).out,
                net(3).in,
                net(5).in):tower_int;
        {component containing spindle "int" of the global state space}

        tower3:(dest,
                net(2).out
                net(5).out
                net(1).in,
                net(6).in):tower_dest;
        {component containing spindle "dest" of the global state
        space}
```

Figure 35. Component Declarations

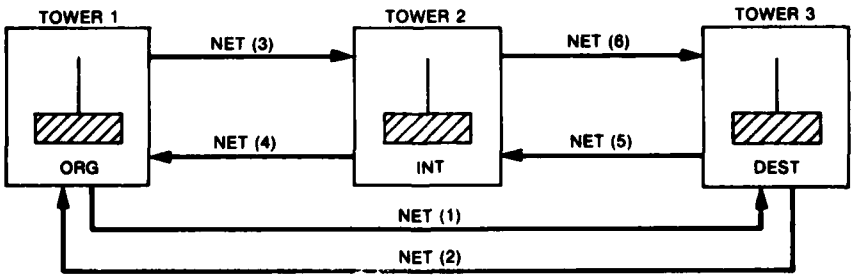


Figure 36. Partition of the Machine "three_towers"

```
communication behavior

function
let c1,c2:obj message channel
let i,j:posint such that INDEX(net,i) and INDEX(net,j) and i≠j
[{at any time only one deal can be in progress}
  forall k,l:posint[if effects(DEALS(net(i).out))<k>
                   precedes effects (DEALS(net(j).out))<l>
                   then effects(gets(net(i).in))<k>
                        precedes effects (DEALS(net(j).out))<l>
]
and
{only "tower3" can issue a stop command}
 forall k:posint[if net(i).out.window.op
                 prior effects (puts(net(i).out))<k>=stop
                 then i=5 or i=2
]
```

Figure 37. Communication Behavior

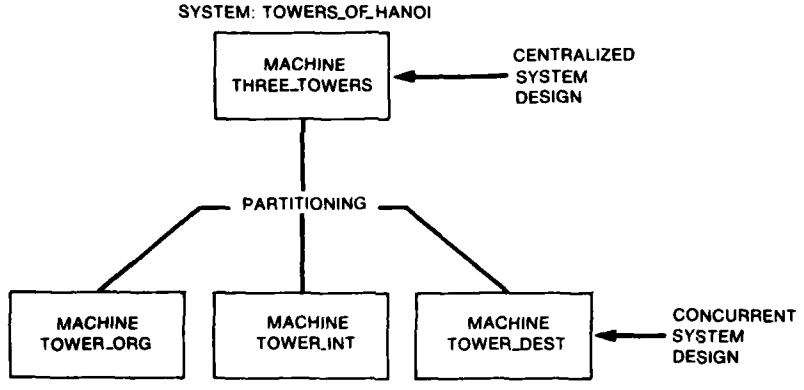


Figure 38. CSDL Structure of the System "towers_of_hanoi"

```
machine tower_org(n:integer)
   declarations
       {declaration of predicates, types, objects
       and actions needed to define the component
       "tower 1" of the top-level design}
   end declarations
       .
       .
       .
end tower_org
```

Figure 39. Outline of the Machine "tower_org"

design. As declared in Figure 35, the component "tower 1" includes a spindle, two message inlets and two message outlets. Following the component declaration for "tower 1" in Figure 35, the object declaration section for the machine "tower_org" has the structure shown in Figure 40.

The design of the data types for the communication structure in Figure 33 suggests the action declarations shown in Figure 39. There are actions for offering a ring for sale (sell) and for closing a sale (close_sale) by sending a ring to a successful bidder or terminating a deal with an unsuccessful bidder. On the other hand, there are actions for making a bid to buy a ring (buy) and for terminating both successful and unsuccessful attempts to buy a ring (close_buy). An additional action is needed for rejecting unwanted bids (reject) received from the other towers.

The two predicates "ORGDUPLEX" and "ORGACCEPT" used in the action declarations of Figure 41 are defined in Figure 42. The predicate "ORGDUPLEX" expresses the fact that the two argument objects are an inlet and an outlet of the machine "tower_org" which are connected to the same machine. The predicate "ORGACCEPT" expresses the condition for bidding for a ring on sale. For this condition to be true, the arithmetic difference between the ring on top of "org" and the ring on sale must be odd, and if there is no ring on the spindle "org", the arithmetic difference between n and the ring on sale must be even (n is the number of rings in the game).

5.2.2 Specifications

The specifications of the states and behavior of the machine "tower_org" are given in Figures 43 and 44 respectively. The states specification refers to the state of the spindle "org" as well as the state of the inlet through which the stop command is received. To demonstrate the derivation of the behavior of specification of the machine "tower_org", the requirements stated in the top-level design are repeated.

Requirements to ensure the game rules:

(1) Any move involves a single ring only.
(2) Any ring to be moved must be on top of a spindle.
(3) No larger ring can be placed on top of a smaller ring.
(4) When a ring is moved, all smaller rings must be on the spindle not involved in the move.

```
objects

    org:tower;
    to_dest:message outlet;
    to_int:message outlet;
    from_int:message inlet;
    from_dest:message inlet;
```

Figure 40. Object Declarations for "tower_org"

```
actions

    sell
       {offer a ring for sale}
       pre [there is at least one ring on the spindle "org"]
       post [send a message of the form "sale,org.high" to both
            the other two components in the partition];

    close_sale (in:message inlet, out: message outlet)
              {after receiving a bid through "in", close the deal
              by sending an appropriate message through "out"}
       pre ORGDUPLEX (in,out) and
           [[a bid for "org.high" has been received] or
           [a bid for a ring other than "org.high" has been received] or
           [there is no ring on spindle "org"]]
       post if [a bid for "org.high" has been received]
           then [[delete the ring on top of "org"] and [send a
                message of the form "deal,org.high" to the bidder]]
           and if [[a bid for a ring other than "org.high" has
                  been received] or [there is no ring on spindle "org"]]
              then [send a "nodeal" message to the bidder];

    buy (in:message inlet, out:message outlet)
        {after receiving an offer of a ring for sale through "in",
        send a bid for this ring through "out"}
       pre ORGDUPLEX (in,out) and ORGACCEPT (in.window.ring) and
           [an offer of a ring for sale has been received]
       post [send a message of the form "bid,in.window.ring"];

    close_buy (in:message inlet, out:message outlet)
             {after having sent a bid through "out", receive a
             positive or negative response through "in"}
       pre ORGDUPLEX (in,out) and in.window.ring=out.window.ring and
           [a bid has been sent through "out"] and
           [[a "deal" message has been received through "in"] or
           [a "nodeal" message has been received]]
       post if [in.window.ring=out.window.ring and
                [a "deal" message has been received]]
            then [add "in.window.ring" to the top of the spindle "org"];

    reject(in1:message inlet, out1:message outlet, in2:message inlet,
          out2:message outlet)
          {after receiving an offer of a ring for sale through "in1"
          and sending a bid for that ring through "out1", all
          bids coming through "in2" are rejected by sending a "nodeal"
          message through "out2", until the original deal is closed}
       pre ORGDUPLEX(in1,out1) and ORGDUPLEX (in2,out2) and
           [a "sale" message has been received through "in1"] and
           [a "bid" message has been sent through "out1"] and
           [a "bid" message has been received through "in2"]
       post [send a "nodeal" message through "out2"];
```

Figure 41. Action Declarations for "tower_org"

```
predicates
   ORGDUPLEX(obj in:message inlet, obj out:message outlet)
       means in=from_int and out=to_int
             or in=from_dest and out=to_dest;

   ORGACCEPT(r:ring_size)
       means if org.dom>0 then [(org.high-r) mod 2=1
            and org.high-r>0]
       and if org.dom=0 then (n-r) mod 2=0;
```

Figure 42. Predicate Declarations for tower_org

```
states

     {specification of the initial state of the spindle "org"}
     init org.dom=n
          and forall i:posint[if INDEX (org,i)
                 then org(i)=org(i-1)-1];

     {specification of the final state of the spindle "org"
     and the communication interface "from_dest"}

     final org.dom = 0
           and from_dest.window.op=stop
```

Figure 43. States Specification of "tower_org"

```
behavior
function

     let in1, in2 :obj message inlet,
          out1, out2 :obj message outlet
     suchthat in1 ≠ in2 and out1 ≠ out2
          and ORGDUPLEX (in1, out1)
          and ORGDUPLEX (in2, out2):

     [{no move must reverse the preceding move}
     last close_sale (in1, out1) later close_sale (in2,out2)
        before first buy (in1, out1)
     or
     last close_sale (in1, out1) later close_buy (in2, out2)
        before first buy (in1, out1)]

     and
     {lock internal state after sending out a bid}
     first buy (in1, out1) later last close_buy (in1, out1)
        before first buy (in2, out2)
     and
     last buy (in1, out1) before reject (in1, out1, in2, out2)
        later first close_buy (in1, out1)];
```

Figure 44. Behavior Specification of "tower_org"

Requirements to ensure computational progress:

(5) At any time, only one ring is being moved.

(6) No move must reverse the preceding move.

(7) Except for the construction of the full tower on "dest", the construction of a subtower with a base of size i must be followed by the move of the ring with size i+1 between the other two spindles.

In the top-level design, requirements (1) through (4) were satisfied by the definition of the action "move" and the type "tower". The design of the machine "tower_org" does not use the action "move". Therefore, these requirements need to be expressed in terms of the declared actions and types of the machine "tower_org".

Requirement (1) is satisfied by the design of the actions of "tower_org". The action "close_sale" moves single rings only (see Figure 41). Requirement (2) is also satisfied by the actions in Figure 41. The action "sale" offers for sale only the ring on top of the spindle, and the action "close_sale" removes the ring on top of the spindle. Requirement (3) is satisfied by the definition of the action "buy". This action issues a bid for a ring on sale only, if the ring on sale is smaller than the ring on top of the spindle (see the definition of ORGACCEPT in Figure 42). Requirement (4) is discussed below. Requirement (5) is satisfied by the specification of the communication behavior (see Figure 37).

To satisfy global requirements, a single machine in a partition can only refer to its internal state including its local inlets and outlets. Thus, the design of a concurrent solution to the Towers of Hanoi game must be based on a mechanism which ensures requirements (4), (6) and (7) on the basis of the local state of an individual machine in the partition.

One condition for such a mechanism is given by the predicate "ORGACCEPT" in Figure 42. Note that this predicate refers to the local state only.

A second condition is given by the fact that after a machine sends out a bid for a ring on sale, it locks its local state. Locking of the local state involves rejecting all bids which come in from the other machines and ignoring all offers of rings for sale received from the other machines.

Requirement (4) is satisfied by the mechanism described above. A ring can only be sold, when it is on top of a spindle. A ring can only be acquired, if all rings on the receiving spindle are larger. If these conditions are satisfied when a deal is initiated, the locking of the internal state of the bidder guarantees that they remain satisfied until the deal is closed. Thus, all rings smaller than the ring to be moved reside on the spindle not involved in the move.

Requirement (6) needs to be restated. To guarantee that the previous move is not reversed, the following must hold. After moving a ring to another machine, the machine "tower_org" must either move another ring to the machine not involved in the first move or accept another ring from that machine, before it can bid on a ring being offered for sale by the destination of the original move. This requirement is formally stated in Figure 44.

Requirement (7) is satisfied by the mechanism described above. The condition defined by "ORGACCEPT" and the locking of the internal state after a bid has been sent out guarantee that in every situation exactly one move is possible. This move satisfies requirement (7) as can be observed from the example in Figure 21.

To ensure the locking of the local state, an additional requirement needs to be specified. This requirement states that, after sending out a bid, no other bid is sent out, and all bids received from other machines are rejected until the original deal is closed. This requirement is also stated formally in Figure 44.

5.2.3 Programs

The structure of a CSDL programs chapter is shown in Figure 45.

The data type declarations for both the top-level design and the machine "tower_org" (see Figure 25) do not yet include definitions of type operations. So far, all functional relationships have been expressed in terms of the declared actions. For the procedural description of the machine "tower_org" operations are needed for the manipulation of the internal state including the spindle "org" and the appropriate inlets and outlets. The operations on inlets and outlets are constructs of CSDL. Thus, the data type "tower" needs to be extended to provide appropriate type operations for the

manipulation of the spindle "org". These operations must allow the top of the spindle to be accessed, the top of the spindle to be deleted, and a ring to be added to the spindle. The extension of the data type "tower" is shown in Figure 46. This declaration of "tower" replaces the original type declaration in both the machine "three_towers" and the machine "tower_org".

The controller of the machine "tower_org" shown in Figure 47 includes three major sections:

- o The variable declarations
- o The initialization of variables and machine objects
- o The actual control algorithm

The outer control loop of "org_controller" has four major pieces, identified by the four guards. When none of the guards is true, i.e., when "terminated" is true, the controller terminates. The first guard is true if the game is not yet terminated, no new messages were received from the other two machines, and the spindle is empty. In this case nothing is to be done, and the controller idles. The second guard is true for the same condition except that there are some rings on the spindle. Thus, the ring on top of the spindle can be offered for sale. To this end, the procedure "sell" is called as described in Figure 48. The third and fourth guards are true if the game is not yet terminated and a new message was received from either of the machines containing the spindles "int" and "dest". The controller texts for dealing with the machines containing the spindles "int" and "dest" are given in Figure 49 and 50, respectively.

These controller texts are similar. In both cases, the messages received may offer a ring for sale, contain a bid for the ring on top of the spindle "org", contain a ring in a successful completion of a deal, or contain a negative acknowledgement for a bid issued earlier. For the communication with the machine containing spindle "dest", the "stop" command may be received in addition to these messages. The controller reacts to these messages as specified in the appropriate action declarations and the behavior specifications. The actions to be performed in the individual cases are presented as procedure calls.

The programs called by the controller are described in Figures 48 and 51 through 54. These programs correspond directly to the actions declared in Figure 41.

```
programs
     program controller
             {description of "controller"}
     end controller

     program name1
             {description of program "name1"}
     end name1
          .
          .
          .
     program name n
             {description of program "name n"}
     end name n

end programs
```

Figure 45. Structure of the Programs Chapter

```
tower:ring_size array
     {a spindle is represented by an array of rings}
     let t:tower
     invariant
          {the rings on a spindle are ordered by their
          size such that no larger ring is on top of a
          smaller ring}
          forall i,j:posint[if INDEX (t,i) and INDEX (t,j)
                    then [if i<j then t(i)>t(j)]
          and t.dom≤n

     vfun top returns ring:ring_size
          {returns the ring on top of the spindle}
          in t.dom>0
          out ring'=t.high

     ofun add (ring:ring_size)
          {adds a ring to the top of the spindle}
          in t.dom <n and t.high>ring or
             t.dom=0
          out t'.hib=t.hib+1 and
              t'.high=ring

     ofun delete
          {deletes the ring on top of the spindle}
          in t.dom>0
          out t'.hib=t.hib-1
end tower;
```

Figure 46. Extension of Type "tower"

```
program org_controller

     variables
     i:integer; {count}
     locked, terminated:boolean; {boolean state indicators}
     int_in,dest_in:message; {intermediate storage of messages}
     last_ring:ring_size; {history variable for recording the ring
                           involved in the last transaction}

     text

     {initialization of the spindle "org"}
     org:=(1);
     i:=1;
     do i<n -> org(i):=n-i+1;i:=i+1 od

     {initialization of the controller variables}
     terminated:=false;
     locked:=false;
     last_ring:=n;

     {control loop of the control algorithm}
     do not terminated and org.dom=0 and not from_int.came
           and not from_dest.came
           {not terminated and no ring for sale and
           no message received}
        ->skip
     [] not terminated and org.dom>0 and not from_int.came
           and not from_dest.came
           {not terminated and no message received,
           but ring for sale}
        ->sell {see Figure 48}
     [] not terminated and from_int.came
        {not terminated and message received from the
        machine containing spindle "int"}
        {see Figure 49}
     [] not terminated and from_dest.came
        {not terminated and message received from the
        machine containing spindle "dest"
             {see Figure 50}
     od

end org_controller;
```

Figure 47. Controller of "tower_org"

```
program sell {offer a ring for sale}

pre org.dom>0 {at least one ring for sale}

post to_dest'.window.op=sale {send a message of the form}
     and to_dest'.window.ring=org.high" {"sale,org.high", to the other}
     and to_int'.window.op=sale {two machines}
     and to_int'.window.ring=org.high; {in the partition}

variables
mess:message;

text

mess.op,mess.ring:=sale,org.top; {generate the message}
to_dest.put(mess); {send to destination}
to_int.put(mess); {send to intermediate}

end sell
```

Figure 48. Program "sell"

```
[] not terminated and from_int.came
    ->int_in:=from_int.get; {save incoming message}
      if int_in.op=sale {ring for sale?}
         ->if locked=true {machine state locked?}
              ->skip {ignore the offer for sale}
           [] locked=false {machine state unlocked?}
              ->if last_ring=int_in.ring {was same ring been involved}
                                          {in previous transaction?}
                   ->skip
                [] last_ring≠int_in.ring
                   ->buy(locked,int_in,to_int) {do "buy"}
                fi
           fi
      [] int_in.op=bid {is a bid coming in?}
         ->if locked=true {machine state locked?}
              ->reject (int_in,to_int) {do "reject"}
           [] locked=false {machine state unlocked?}
              ->close_sale(last_ring,int_in,to_int) {do "close_sale"}
           fi
      [] int_in.op=deal {is a ring coming in}
         ->close_buy(locked,last_ring,int_in)
                                          {do "close_buy"}
      [] int_in.op=nodeal {was a bid rejected}
         ->locked:=false
      fi
           .
           .
           .
```

Figure 49. Dealing with Machine Containing "int"

```
[] not terminated and from_dest.came
    ->dest_in:=from_dest.get; {save in:=omin:= message}
      if dest_in.op=stop {"stop" command received?}
         ->terminated:=true {set termination indicator}
      [] dest_in.op=sale {ring for sale?}
         ->if locked=true {machine state locked?}
              ->skip {ignore offer for sale}
           [] locked=false {machine state unlocked?}
              ->if last_ring=dest_in.ring {was same ring involved}
                                          {in previous transaction?}
                   ->skip
                [] last_ring≠dest_in.ring
                   ->buy(locked,dest_in,to_dest) {do "buy"}
                fi
           fi
      [] dest_in.op=bid {bid coming in?}
         ->if locked=true {machine state locked?}
              ->reject(dest_in,to_dest) {do "reject"}
           [] locked=false
              ->close_sale(last_ring,dest_in,to_dest) {do "close_sale"}
           fi
      [] dest_in.op=deal {ring coming in?}
         ->close_buy(locked,last_ring,dest_in) {do "close_buy"}
      [] dest_in.op=nodeal {bid rejected?}
         ->locked:=false {unlock machine state}
      fi
         .
         .
         .
```

Figure 50. Dealing with Machine Containing "dest"

```
program buy(locked:boolean,in:message,var out:message outlet)
                                          {bid for ring on sale}

pre ORGACCEPT(in.ring); {the ring must be aceptable}

post out'.window.op=bid {send a message of the}
     and out'.window.ring=in.ring {form "bid,in.ring" and}
     and locked'=true; {lock the machine state}

variables
mess:message; {message variable}

text

if (org.dom>0 and (org.hib-in.ring)mod2=1) {test acceptance of the}
   or (org.dom=0 and (n-in.ring)mod2=0) {ring for sale}
   ->mess.op,mess.ring:=bid,in.ring; {generate the message}
     locked:=true; {lock the machine state}
     out.put(mess) {send the message}
[] (org.dom>0 and (org.hib-in.ring)mod2=0) {test non-acceptance}
  or (org.dom=0 and (n - in.ring)mod2=1) {of the ring for sale}
  ->skip {ignore the offer}
fi

end buy
```

Figure 51. Program "buy"

```
program reject(in:message,out:message outlet) {reject bids while the
                                                machine state is locked}

pre true;

post out'.window.op=nodeal {send a "nodeal" message}
     and out'.window.ring=in.ring;

variables
mess:message;

text

mess:op,mess.ring:=nodeal,in.ring; {create message}
out.put(mess); {send message}

end reject
```

Figure 52. Program "reject"

```
program close_sale(last_ring:ring_size,in:message,out:message outlet)
                                                {close a deal}

pre in.ring=org.high {a bid for top ring}
                                                {"org" has been received}
    or in.ring=org.high or org.dom=0; {a bid for ring in middle}
                                                {of "org" has been received}

post if in.ring=org.high
     then [org'.hib=org.hib-1 {delete top of "org",}
          and out'.window.op=deal {send a message of the}
          and out'.window.ring=org.high {form "deal,org.high"}
          and last_ring'=org.high] {to the bidder}
     and if [in.ring≠org.high or org.dom=0]
         then [out'.window.op=nodeal {send a "nodeal"message}
              and out'.window.ring=in.ring]; {to the bidder}

variables
mess:message:

text

if in.ring=org.top and org.dom>0 {bid acceptable}
   ->mess.op,mess.ring:=deal,org.top; {create message}
     last_ring:=org.top; {remember top ring}
     org.delete; {delete top ring}
     out.put(mess) {and send it}
[] in.ring≠org.top or org.dom=0 {bid unacceptable}
   ->mess.op,mess.ring:=nodeal,int_in.ring; {create message}
     out.put(mess) {and send it}
fi

end close_sale
```

Figure 53. Program "close_sale"

```
program close_buy(locked:boolean,last_ring:ring_size,in:message)
                                                {receive a bid response}

pre true;

post org'.hib=org=org.high+1 {add new ring}
     and org'.high=in.ring {to the top of "org"}
     and locked'=false {unlock machine state}
     and last_ring'=in.ring; {record new ring}

variables
mess:message;

text
org.add(in.ring); {add new ring to top of "org"}
locked:=false; {unlock machine state}
last_ring:=in.ring {record new ring}

end close_buy
```

Figure 54. Program "close_buy"

5.2.4 Remaining Machines

To complete the design of the system "towers_of_hanoi", the two machines containing the spindles "int" and "dest", respectively, would have to be defined. These machines would define the components "tower2" and "tower3" in the partition of the machine "three_towers" (see Figure 35). Given the definition of these two machines, the document for the definition of the system "towers_of_hanoi" has the overall structure shown in Figure 55.

Except for minor differences, the machines in the partition of "three_towers" are identical. Therefore, the complete design of the machines "tower_int" and "tower_dest" is omitted. The definition of these machines follows the principles illustrated for the definition of the machine "tower_org". The following differences would have to be observed in the design of these two machines:

tower_int:

- In the initial state, the spindle "int" is empty.
- The acceptance criteria for rings on sale needs to be defined such that for int.dom=0, the arithmetical difference between the size of the ring to be accepted and n is odd. This is equivalent to assuming an imaginary ring of size n at the bottom of "int".

tower_dest:

- In the initial state, the spindle "dest" is empty.
- In the final state, the spindle "dest" contains the full tower.
- The termination of the game has to be determined by detecting that the full tower has been constructed on "dest". Upon termination the "stop" command needs to be issued to the machines "tower_org" and "tower_int".

5.3 DESIGN OF A COMMUNICATION SUBSYSTEM

The solution to the Towers of Hanoi problem presented above was made up of three concurrent machines communicating among themselves over six point-to-point channels. This design separates concern for the required logical communication from

```
system towers_of_hanoi
      {list of machines contained in the definition}
      three_towers(n);
      tower_org(n);
      tower_int(n);
      tower_dest(n);

      machine three_towers(n)
         {definition of "three_towers"}
      end three_towers;

      machine tower_org(n)
         {definition of "tower_org"}
      end tower_org;

      machine tower_int(n)
         {definition of "tower_int"}
      end tower_int;

      machine tower_dest(n)
         {definition of "tower_dest"}
      end tower_dest;

end towers_of_hanoi
```

Figure 55. Overall Structure of "towers_of_hanoi"

concern with supplying that communication capability. That set of channels will now be refined into a central communication subsystem. This refinement is accomplished by means of the CSDL type refinement mechanism. The "com_struc" data type will be refined into a single communication subnet which satisfies the requirements posed at the upper level. The refinement is developed in the design of machine network, which refines com_strue.

5.3.1 Refining Machine Description

The major design decisions made at this level are to frame messages with source and destination addresses and to use a Y-shaped message switch, each arm of which has a unique id, to replace the delta-shaped set of channels. The complexity of system-wide serialization of traffic, required for a direct implementation with a bus for example, will be postponed to another level. Figure 56 shows the logical structure of the "com_struc" data type and how it will be mapped onto the message switch.

The structure of the refining machine "network," then, is three partitions connected by the message switch. Each partition is an interface unit which implements the operations "put", "get", and "came" for its portion of "com_struc". Since the "went" operation is never used and is in fact superfluous because of the requirement on com_struc that the leaves event is never delayed, it is not implemented in any partition.

The type definitions for port addresses, framed messages, and the message switch are shown in Figure 57. The invariant specification of the message switch type includes its port address assignments and the properties of orderly, reliable message routing. To aid in expressing these properties, the predicates "GOES_OUT" and "UNCHANGED" are used; their definitions are given in Figure 58.

The specifications for the "network" machine include the mappings between structural components of the "com_struc" data type and the message switch; these mappings are shown in Figure 59.

The single interface object of type message switch is named "switch", and the three interface units are "iu_A", "iu_B", and "iu_C"; these declarations are shown in

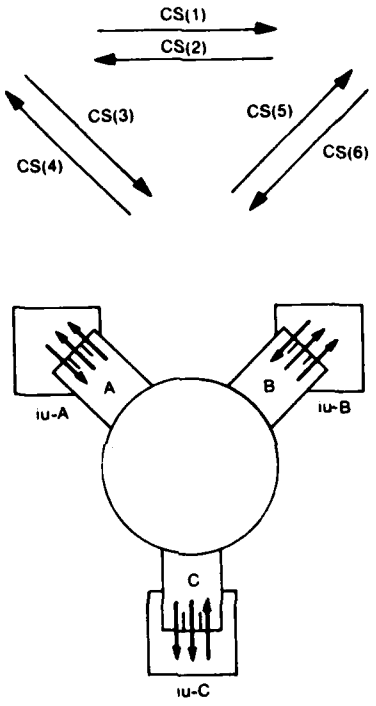


Figure 56. Mapping Between

```
types
port_id: (A [] B [] C);
net_frame: (from,to: port_id,body: message);
mail_box: (in1,in2: net_frame inlet, out: net_frame outlet,id: port_id);
message_switch: (endA,endB,endC: mail_box)
   let ms: message_switch
   inv {global unique address for each port}
       ms.endA.id=A & ms.endB.id=B & ms.endC.id=C
   and {"from" field is always set to port_id of sender}
   forall i: posint [
      [ms.endA.out.window.from prior effects (leaves(ms.endA.out))<i>=A]
      and
      [ms.endB.out.window.from prior effects (leaves(ms.endB.out))<i>=B]
      and
      [ms.endC.out.window.from prior effects (leaves(ms.endC.out))<i>=C]
   ]
   and {every message gets to its destination unchanged}
   forall i: posint [
      [effects (GOES_OUT(ms.endA,B))<i>
         later effects (arrives(ms.endB.in1))<i>
      and UNCHANGED(ms.endA,B,ms.endB.in1)]
      and
      [effects (GOES_OUTms.endA,C))<i>
         later effects(arrives(ms.endC.in1))<i>
      and UNCHANGED(ms.endA,C,ms.endC.in1,i)]
      and
      [effects (GOES_OUT(ms.endB,A))<i>
         later effects (arrives(ms.endA.in1))<i>
      and UNCHANGED(ms.endB,A,ms.endA.in1,i)]
      and
      [effects (GOES_OUT(ms.endB,C))<i>
         later effects (arrives(ms.endC.in2))<i>
      and UNCHANGED(ms.endB,Cms.endC.in1,i)]
      and
      [effects (GOES_OUT(ms.endC,A))<i>
         later effects (arrives(ms.endA.in2))<i>
      and UNCHANGED(ms.endC,A,ms.endA.in2,i)]
      and
      [effects (GOES_OUT(ms.endC,B))<i>
         later effects(arrives(ms.endB.in2))<i>
      and UNCHANGED(ms.endC,B,ms.endB.in2,i)
   ]
```

Figure 57. Required Types "com_struc" and Message Switch

```
predicates
GOES_OUT(obj from: mail_box, to: port_id) means
   leaves(from.out) and from.out.window.to=to;

UNCHANGED(obj from: mail_box,tid: port_id,
          obj to: net_frame inlet, i: posint) means
   {no messages are altered in transit}
   from.out.window prior effects (GOES_OUT(from,tid))<i>
      = to.window after effects (arrives(to))<i>;
```

Figure 58. Predicates

```
let cs: com_struc
switch.endA.out represents cs(1).out;
switch.endA.out represents cs(3).out;
switch.endA.in1 represents cs(2).in;
switch.endA.in2 represents cs(4).in;

switch.endB.out represents cs(2).out;
switch.endB.out represents cs(6).out;
switch.endB.in1 represents cs(1).in;
switch.endB.in2 represents cs(5).in;

switch.endC.out represents cs(4).out;
switch.endC.out represents cs(5).out;
switch.endC.in1 represents cs(3).in;
switch.endC.in2 represents cs(6).in;
```

Figure 59. Mapping Function Between "switch" and "com_struc"

Figure 60. The component declarations indicate that the components are instances of the machine type "interface_unit", differing only in the values of certain system parameters. The switch component names appearing in parenthesis immediately following the individual component names indicate the association between each component and the message switch. This constitutes a textual method of declaring the system interconnection topology.

With the declaration of the partition chapter the description of the network machine is completed. The decision to use a common communication subnet to route messages tagged with sender and destination addresses has been established. It now remains to describe the components at each end of the net which must supply the functions used at the level above.

5.3.2 Partition Machine Description

The declaration

```
machine interface_unit(T1,T2: port_id)
```

introduces the machine description for these components. The parameters are values which particularize various components. These identifier parameters determine the sources various components will be receiving messages from. This machine has one object, mbx, of type mail_box. Programs will be defined which implement various upper level operations; the program - operation associations must be specified. The mailbox in an interface unit represents two channel outlets and two channel inlets, as indicated in Figure 59; this information is used to specify the operation mappings in Figure 61.

The program "send" implements the "put" operation on outlets. The view taken here is that the upper level requirement that it is never necessary to check the flag before doing a "put" operation means that the application system need never be faster than the communications; accordingly, the program "send" waits on the subnet flag before doing a "put". The result is that the application can be delayed by the "put" operations which it uses, but such delay is invisible to the application. The "send" program is described in Figure 62. The programs "receive" and "check" implement the "get" and "came" operations respectively. Their descriptions are shown in Figure 63.

With the definitions of these programs, the definition of the first level of refinement of the virtual point-to-point communications system is complete. Plausible subsequent refinements might be 1) to demultiplex the inlet streams and 2) to map onto a single bus through queueing and token passing for resolving contention.

```
partition
interfaces
switch: message_switch;

components
in_A(switch.endA): interface_unit(B,C);
in_B(switch.endB): interface_unit(A,C);
in_C(switch.endC): interface_unit(A,B);

end partition
```

Figure 60. Partition Definition of Machine "network"

```
mappings
let cout1,cout2: message outlet, cin1,cin2: message inlet
    send(I1) represents cout1.put;
    send(I2) represents cout2.put;
    if mbx.in1 represents cin1
       then receive (I1) represents cin1.get;
    if mbx.in1 represents cin1
       then check (I1) represents cin1.came;
    if mbx.in2 represents cin2
       then receive (I2) represents cin2.get;
    if mbx.in2 represents cin2
       then check (I2) represents cin2.came;
```

Figure 61. Operation Mappings

```
program send(id: port_id, ms:message)
pre id=I1 or id=I2
post let f:net_frame
       such that f.from=mbx.id and f.to=id and f.body=ms
     [mbx'.out.window=f and mbx'.out.flag=false]

variables f: net_frame

text
f.from,f.to,f.body := mbx.id,id,ms;
{wait till ready} do not mbx.out.went -> skip od;
mbx.out.put(f)

end send;
```

Figure 62. The Program SEND

```
program receive(id: port_id) returns ms: message
pre id=I1 or id=I2
post [if id=I1 then ms'=mbx.in1.window and mbx'.in1.flag=false]
       and [if id=I2 then ms'=mbx.in2.window and mbx'.in2.flag=false]

text
if id=I1 -> ms := mbx.in1.get
[] id=I2 -> ms := mbx.in2.get
fi

end receive

program check(id: port_id) returns b:boolean
pre id=I1 or id=I2
post [if id=I1 then b'=mbx.in1.flag]
       and [if id=I2 then b'=mbx.in2.flag]

text
if id=I1 -> b := mbx.in1.came
[] id=I2 -> b := mbx.in2.came
fi

end check;
```

Figure 63. The Programs RECEIVE and CHECK

SECTION 6
PERFORMANCE SPECIFICATION IN CSDL

In this section, a formalism is presented which supports performance analysis of system designs defined in CSDL. This formalism allows performance constraints to be expressed within CSDL system definition documents and input parameters as well as measurements to be obtained from a system performance analysis to be extracted from such a document.

6.1 OBJECTIVES

Performance requirements define some of the most important properties a system must possess. Failure of a system design to meet given performance requirements necessitates modification of the design or even redesign of the system. Therefore, it is essential that system definitions include information about performance requirements in order to support the designer in analysing system designs and design decisions. Responding to this need, one of the objectives is to incorporate concepts, constructs and operations into the CSDL structure to allow the designer to:

- o Make statements about system performance
- o Ask questions about system performance, during system design

These statements and questions must be applicable and meaningful at all levels of system definition, i.e., at all levels of refinement.

Minor extensions to the CSDL computational model and the introduction of a small set of new constructs as discussed in 6.2 and 6.3, are required to achieve these goals. Before proceeding to those details, however, this subsection presents an overview of performance constructs.

6.1.1 Statements, Questions and Questments

Many performance questions and statements relate to duration or occurrence of an epoch for some activity A. Thus a designer may want to

state A takes t time units ! or

ask if A takes t time units ?

where "!" denotes a statement and "?" denotes a question. The statement conveys information about properties that A must possess. The question asks about A's relationship with respect to a certain property. Not all properties relate to time, and common questions and statements may be classified as logical or numeric:

A precedes B ! (a logical statement i.e., a behavior assertion)
A precedes B ? (a logical question)

A takes a time 0 < t < n ! (numeric statement)
A takes a time 0 < t < n ? (logical question)
A takes a time t = ? (a numeric question)

To reiterate, statements convey information about properties that must or will hold and as such are true assertions. The validity of logical questions is open; i.e., unknown, as are parameters of numeric questions.

The purpose of questions is to indicate properties about which information is desired or required. Information can only be acquired by some form of performance analysis. Analysis can be achieved by:

- Performing static algebraic manipulations on statements [BOOT79], [WEGB76], [YAU81], [FRAN79]
- Mapping the system definition into a dynamic queueing network representation, and then solving the network [DENN78], [GELE80], [RAMA80], [SAUE79]
- Mapping the system definition into a dynamic simulator and performing simulations [SANG79], [CAVO81], [FRAN77]
- Realizing the system or a simplified prototype and collecting operational data from the running system [KAIN79], [BERG80]

All four approaches are relevant, and linkages between CSDL system definitions and system performance analysis are discussed in 6.5.

Here the concern remains with a qualitative assessment of the descriptive needs of CSDL in order to address performance issues, assuming for the moment the existence of any required link to a performance analysis tool. It is necessary to distinguish between statements that are valid and those that should (must) be valid, but are not obviously seen to be so. Determination of the validity of statements in this latter category remains an open question, and here they are referred to as questments. Questments must be resolved by performance analysis, and are denoted by the !? pair. As an illustration consider:

The average throughput of A is X ! (numeric statement)
The mean throughput of A is ? (numeric question)
The mean throughput of A is X !? (a questment)

Determination of the invalidity of a questment constitutes a system performance violation and necessitates modification of the system definition or even redesign. The inability to validate or invalidate a questment is not considered.

6.1.2 Time and Counts

Many of the preceding examples relate to time and therefore imply an ability to record the passage of time. One of the prime attributes of some concurrent systems (in particular loosely coupled systems) is near autonomy of the system components. The individual components of such systems adhere to individual clocks. While such systems may require multiple clocks for operation, one global clock is desirable and sufficient for use in performance statements, questions, questments, and finally is necessary for performance analyses. Moreover, proper use of the single clock for performance purposes does not invalidate or disturb the inherent multiple clock nature of certain systems [KAIN79].

The performance approach therefore assumes the presence of a single globally available clock against which the passage of time can be measured. The basic use of the clock will be to record intervals and epochs (i.e., time points) at which events (described in 6.2) occur. Here an event is loosely associated with the occurrence of an activity or, if the activity consumes time (i.e., is not instantaneous) with its initiation or completion.

The utility of a clock is much enhanced by an ability to count (record) the number of occurrences of an event or activity and to refer to a given occcurrence, say the i-th, in the time ordered sequence of occurrences. As shall be seen in 6.3, CSDL already has sufficient capability to deal with counts.

6.1.3 Activities and CSDL Definitions

CSDL admits four possible hierarchically ordered levels of system description:

- o System level (the system is viewed as a black box)
- o Action level (individual activities are viewed as black boxes)
- o Partitioned level (actions are dispersed among machines and communications is considered)
- o Program level (action specifications are replaced by procedural programs)

Each level in the hierarchy contains more information and detail than preceding levels. Each level has associated activities and related performance issues:

- o The system level has the atomic system activity, with determination of system throughput as a possible measure.
- o The action level has actions and type function invocation activities with action execution times as possible measures.
- o The partitioned level has actions, types and channel activities, and transmission times as possible measures.
- o The program level has actions, types, channel, and program statement activities and any and all measures related to those entities.

Each activity is a likely candidate for involvement in a performance statement, question, or questment.

6.1.4 An Experimental Framework

For dynamic system analysis related to statements, questions and questments two experimental frameworks are possible. One is theoretical and implies observation of system behavior over an infinite time. Such is the assumption employed in most queueing theoretic results. The other is operational and implies observation of the system for a finite period of time.

It is believed that most CSDL definitions will require dynamic analysis, which will require simulation. The tenants of operational analysis seem adequate for most such analyses, and additionally have a relationship to queueing theory [BUZE76], [DENN78]. Under operational analysis a record is made of:

T - Observation period
A - Number of requests (activations, users arrivals) during T for a given activity
B - Time ($B \leq T$) that the activity is engaged (present) during T
C - Number of occurrences (uses) of the activity completed during T
W - Backlog time integral associated with T, where backlog N = A - C

From these observed operational measures we can derive:

(1) A/T request rate (mean)
(2) C/T service rate (mean)
(3) B/T utilization (fraction of time busy)
(4) B/C the mean service time

In subsequent subsections these notions are made concrete, first by extending the computational model to allow for a clock, then by introducing CSDL constructs to allow for performance statements, questions, and questments about system performance. The number of new constructs required to characterize performance is minimal. The power to characterize performance already exists within CSDL. The extensions to CSDL needed to support the experimental framework described above are the global clock and the ability to characterize random variables which are often associated with activity times and counts in dynamic systems.

6.2 EXTENSION TO THE CSDL COMPUTATIONAL MODEL

To facilitate the specification of system performance in CSDL definitions, a single extension to the CSDL computational model is necessary. As described in 6.1, there exists a need for a mechanism to relate the passage of time during execution of the system to the time spent with the pure information processing. This need can be satisfied by introducing a single, global clock for performance purposes. This clock can be used to specify a required passage of time among the many activities in a concurrent system. However, an implementation of a system may not refer to such a clock. Rather each autonomous element of the actual system has its own clock, with respect to which proper adherence to performance specifications must be satisfied. Thus, use of a global clock isolates the expression of performance specifications from the means by which the performance specifications are satisfied.

The extension that is added to the CSDL computational model is the ability to specify an absolute time for any event in a system execution. As discussed in 4.1, an event is a change in value of an object in the object space of a machine. Individual, sets, or sequences of events can be referred to by the specification language. A new language primitive called clock allows statements in the specification language to refer to absolute times of occurrences of individual events. Multiple uses of this primitive allow reference to be made to sets or sequences of events. Thus, more than mere temporal relations among events can be specified. For example, the passage of time between two events can now be expressed by the difference between the absolute times, as given by clock, for each event. Subsection 6.3 defines the specification language primitive clock and gives several examples of performance specifications.

Absolute time is expressed in convenient time units, e.g., seconds. Although not necessary, an arbitrary event at the beginning of a sequence can be designated as occurring at time 0. However, for virtually all performance specifications, relative times, or differences in absolute time, are used.

- Terminating sequences of events can be assumed to start with an initial event at time = 0, and terminate with an event at a future time.

- o Nonterminating sequences of events are assumed to require a non-zero amount of time between each event pair, and thus relative time differences can be used to express performance.

Figure 64 is example of an event history in a machine. The computational model permits temporal assertions to be made that can specify that certain events occur before or after certain other events. Thus, the assertion language can be used to specify that event E1 occurs before event E2, and event E2 occurs before event E3, as indicated in Figure 64. The extension to the computational model allows for an association between events and time. As shown in the figure, event E1 can be specified to occur at a particular time (wih respect to a system clock), and likewise for the other events. Thus, performance information related to delays between occurrences of events and frequencies of events can now be specified.

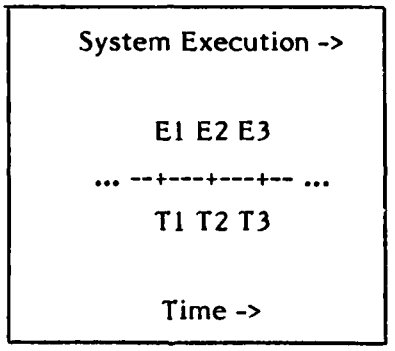


Figure 64. Events and Times of Events

6.3 NOTATION FOR PERFORMANCE SPECIFICATION

This subsection details new constructs and extensions to CSDL that are necessary to specify system performance.

6.3.1 Time

The syntactic construct to express time in the specification language is

clock(E)

where E is an expression identifying an event (see 4.2); "clock(E)" is a real-valued function and has as its value the time at which the event E occurs. Clock expressions can be used and intermixed with other assertions, as will be shown in later examples.

It is often desirable to specify that an interval of time or number of occurrences of an activity are within a range of possible time values for any behavior of the system. These specifications are expressed as follows: Let E1 and E2 be two events of interest. Then t = clock(E1) - clock(E2) defines the time between the occurrence of the two events, and $t1 \leq t \leq t2$ expresses the fact that T is in the range t1 to t2.

6.3.2 Number of Occurrences of Events

Suppose E is an event, for example "arrives(a)". Then "E<i>" refers to the i-th occurrence of the event E. In this example, arrives(a)<i> is the event that characterises the i-th arrival of a message at the inlet a. Thus, a means exists to identify and count occurrences of events as is required by operational analysis.

Additionally, the total number of occurrences of an event E during an observation period is given by the expression

```
#i:posint[E<i>]
```

which denotes the cardinality of the number of occurrences of the event E.

6.3.3 Specification of Stochastic Intervals

It is often desirable to express that a time t has a stochastic, or probabilistic, nature, such that, for example, a time interval between two events will have a clock value randomly drawn from a particular probabilistic distribution. The CSDL construct dist is introduced to allow this kind of performance specification. For example

```
t = dist(negexp,y)
```

means that values of t are assumed to have a negative exponential distribution that has a mean of y. Other distributional forms can be specified by replacing negexp and y

by the name and descriptive parameters of other desired distributions. Tabular distributions may be specified by lists of (value, probability) pairs, in a manner analagous to the term definitions for n and t that appear in the next subsection.

6.3.4 Specification Language Extensions

As indicated in 6.1 the specification language is to be extended to account for terms that are used to form performance statements, questions, and questments. The construct which is used for the integration of these extensions into the specification language is the means construct, which provides for the specification, parameterization and naming of predicates. As an example we could have

p(a,b) means a < b

which (as a predicate) has a logical value, i.e., is true or false.

The extension for performance statement purposes provides for terms, i.e., functional definitions, such as

p(a,b):real means a + b

which has a real numeric value rather than a logical value. Formally, these constructs specify terms, and the right-hand sides may be arithmetic or temporal terms possibly embedded in the context of a let clause. As the specification language is tentative and not fully detailed in this report, syntax diagrams are not provided for the extended means construct.

As an example of the extension, consider specification of the epochs at which changes in the number of elements in an array, a, occur, as well as specification of the number of elements the array contains following the changes. Given the extension these values are specified by

n(a:integer array, i:posint):integer means
a.dom aftereffects(a'.dom ≠ a.dom)<i>
t(a:integer array, i:posint):real means
clock(effects(a'.dom ≠ a.dom)<i>)

Furthermore the time the array remains in a given state after a change is specified by a term of the form

```
t1(a:integer array, i:posint):real means
 clock(effects(a'.dom ≠ a.dom)<i>) - clock(effects(a'.dom ≠ a.dom)<i-1>)
```

These forms may be thought of as defining numeric functions or sequences of numeric values; either view is consistent with the extension. The extension is sufficient to allow expression of the performance measures used in operational analysis, including discrete time integrals and snapshots. Two examples are given to elaborate this point.

Consider first a desire to snapshot the virtual buffer contents of a channel at time t. The snapshot specification is

```
size(a:m channel, t:real):integer means
#i:posint[clock(effects(leaves(a))<i>) ≤ t] - #i:posint[clock(effects(arrives(a))<i>) ≤ t]
```

Consider second a discrete time integral which is defined as a sum of products. The crucial element of a time integral specification is the sum

```
sum(s:real,x:real array,i:integer) means
  x.lob ≤ i and
  if i > x.hib then s = 0 and
  if i ≤ x.hib then let s1:real such that sum(s1,x,1+i) [s = s1 + x(i)]
```

Then, using sum the time integral of an array a becomes:

```
time_integral(a:real array):real means
let max:integer such that
  max = #y,x:integer[ ⊣ i:integer[x=i and y=n(a,i)]],
  z:integer array, s1:real such that
  sum(s1,z,z.lob) and
  ∀i:integer [if 1 ≤ i ≤ max-1 then z(i) = t1(a,i) * n(a,i)]
[s1]
```

Other more elaborate examples follow in 6.3.6.

6.3.5 Performance Statement Placement

Performance assertions (predicates), which are constructed from arithmetic and temporal terms, may appear in a variety of places in a system description. Specifically they may appear in:

- Specifications Chapter - Within the behavior specifications section a new subsection perf for performance assertions is added,
- Program Definition Text - Performance specifications for sequential programs appear in a perf assertions section, obviating the need for performance specification fields within the actual design text,
- Type Declarations - A performance assertions section is added to the specifications of operations and functions,
- Action Declarations - A performance assertions section is added to the pre and post assertions of action specifications.

Their form parallels those of other sections, as the example of 6.4 illustrates.

6.3.6 Sample Operational Analysis Term Specifications

Before proceeding to an example containing a variety of performance assertions, some useful and simple performance specifications are given which define various performance measures commonly used in assertions.

Departure Rate of Entities from an Outlet -- This term determines the number of messages of type m that leave an outlet of a machine during an observation period, and divides this number by the time interval of the observation period to determine the departure rate, or frequency.

```
Outlet_Departure_Rate(out:m outlet):real means
let last:posint such that
  last=#i:integer [leaves(out)<i>]
[last/(clock(leaves(out)<last>)-clock(leaves(out)<1>))]
```

Departure Rate of Entities from a FIFO Queue -- This specification describes the rate at which entities leave a First-In-First-Out queue. It is assumed that the structure of the queue data type is defined elsewhere and that an array of type x is used to store the elements of the queue.

```
FIFO_Queue_Departure_Rate (A:x array):real means
{A.dom is the size of the array at any point in time. An enqueue into the queue
increments A.dom by one, and a dequeue, or departure, from the queue,
decrements A.dom by one. Therefore, we are interested in the number of times
A.dom is decremented in the observed time interval. Such an event is represented
by effects(A'.dom < A.dom), indicating the new value of A.dom is less than the old
value.}
let last:posint such that
  last=#i:integer [effects(A'dom < A.dom)<1>]
[last/
  (clock(effects(A'dom < A.dom) <lasts>) -
    clock(effects(A'dom < A.dom)<1>))]
```

Average Queue Size for a Queue -- Assume for a queue the same type structure that was described in a previous example. A history of A.dom, giving the number of elements in the queue, might appear as shown in Figure 65.

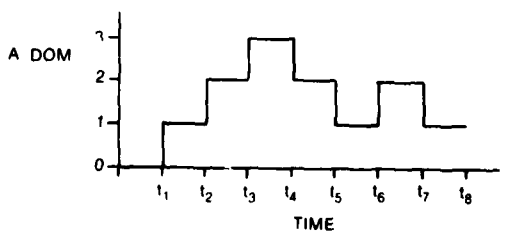


Figure 65. Observation of Queue Size

At time 0 there are no elements in the queue. At time t1 an enqueue is performed, and A.dom is then equal to 1. Also at times t2, t3, and t6 there are enqueues and A.dom is incremented. At times t4, t5, t7 and t8 dequeues are done, and A.dom is decremented each time.

The average queue size is defined to be the area under the curve, which can be obtained from a time integral of A.dom, divided by the time interval t8-t1.

To obtain the average queue size, first define a term to refer to the number of elements in the queue at the i-th time an enqueue or dequeue was performed.

```
n(A:integer array,i:posint):integer means
 A.dom after effects(A'.dom ≠ A.dom)<i>
```

Then define a term to refer to the time at which the i-th enqueue or dequeue occurs.

```
t(A:integer array,i:integer):real means
 clock(effects(A'.dom ≠ A.dom)<i>)
```

Finally, define a term for the average queue size of a FIFO queue, using the two terms given above.

```
FIFO_Average_Queue_Size(A:integer array):real means
let max,s1:integer,z:integer array such that
 {max is the number of enqueues and dequeues}
 max = #x,y:integer[ ] i:integer[y=i and x=n(A,i)]] and
 {s1 is the sum of the elements in the array z.}
 sum(s1,z,z.lob) and
 {Each element in z is the queue size multiplied by the period of time
 during which the queue has that size at the i-th enqueue or dequeue.}
 ∀i:integer[if 1 ≤ i < max then z(i) = n(A,i) * (t(A,i+1)-T(A,i))]
{The average queue size is the sum s1 divided by the observed time.}
[s1 / (t(A,max) - t(A,1))]
```

Variance of Queue Size for a FIFO Queue -- Taking the same approach as in the average queue size example, the variance of queue size is given as:

```
FIFO_Queue_Size_Variance(A:integer array):real means
 {The terms n and t given with the FIFO_Average_Queue_Size
 term are used. For the variance each queue size is squared.}
let max,s1:integer,z:integer array such that
  max = #x,y:integer[ ] i:integer[y=i and x=n(A,i)]] and
  sum(s1,z,z.lob) and
  ∀i:integer[if 1 ≤ i < max then
  z(i) = (n(A,i) * (t(A,i+1) - t(A,i))) * (n(A,i) * (t(A,i+1) - t(A,i)))]
[s1 / (t(A,max) - t(A,1)) - FIFO_Average_Queue_Size(A) *
   FIFO_Average_Queue_Size(A)]
```

Average Waiting Time of Entities in a FIFO Queue -- The average waiting time of entities in a FIFO queue can be obtained by use of Little's Law:

```
Queue_Average_Waiting_Time (A:integer array):real means
   FIFO_Average_Queue_Size(A) / Queue_Departure_Rate(A)
```

Utilization of a FIFO Queue -- Utilization of a queue is defined as the relative fraction of time the queue contains one or more entities. Therefore, when A.dom>0 the queue is non-empty, otherwise it is empty. Associating a new term nv(A,i)=1 with the queue being non-empty at the ith time, and nv(A,i)=0 with it being empty, the utilization is a time integral of the term nv.

```
nv(A:integer array,i:integer):integer means
  if A.dom after effects(A'.dom ≠ A.dom)<i> > 0 then 1 and
  if A.dom after effects(A'.dom ≠ A.dom)<i> = 0 then 0

FIFO_Queue_Utilization(A:integer array):real means
 let max,s1:integer,z:integer array such that
    max = x,y:integer[ ] i:integer[y=i and x=nv(i)]] and
    sum(s1,z,z.lob) and
    ∀i:integer[if 1 ≤ i < max then z(i) = nv(A,i) * (t(A,i+1) - t(A,i))]
 [s1 / (t(A,max)-t(A,1))]
```

Average Queue Size for a Buffered Channel -- Assume a buffered communication channel, A, for which the average buffer length is desired. To obtain the average buffer size, the term t_change is first defined.

```
t_change(A:t channel,i:posint):real means
 clock(effects(leaves(A.in) or arrives(A.out))<i>)
```

Then the average buffer length term is specified:

```
Mean_Buffer_Size(A:t channel):real means
 let max:integer,z:real array,s:real such that
    max = #y,x:integer[ ] i:integer[x=i and y=t_change(A,i)]] and
    Vi:posint[if i ≤ max then
    z(i) = size(A,t_change(A,i)) * (t_change(A,i)-t_change(A,i-1)] and
    sum(s,z,z.lob)
 [s / (t_change(A,max) - t_change(A,1))]
```

6.4 AN ILLUSTRATION OF PERFORMANCE SPECIFICATION IN CSDL

This subsection illustrates the use of (1) the CSDL performance constructs and (2) the performance terms described in 6.3. The system design is an example of a high-level specification of an interactive multiprogrammed computer system. The specification follows the CSDL documentation format and syntax, as given in Section 4 and demonstrates how performance specifications can be integrated into a system specification document. Attention is given primarily to the system's structure and to performance specifications. Less emphasis is given to functional specifications, but comments are used to provide sufficient functional detail.

The interactive computer system we have chosen to illustrate is widely known and has received considerable attention from a performance modeling and analysis viewpoint [DENN78]. CSDL, as will be seen, allows for the specification of at least as much performance information as appears in performance models of such a system. Therefore, it is expected that sufficient performance information can be extracted

from the CSDL description document to build and analyse a performance model. Furthermore, the CSDL document can be used to unambiguously describe the activities in the system, so that the model can be precisely defined.

The interactive computer system model is illustrated in Figure 66. An informal description of the structural, functional, and performance characteristics is given below.

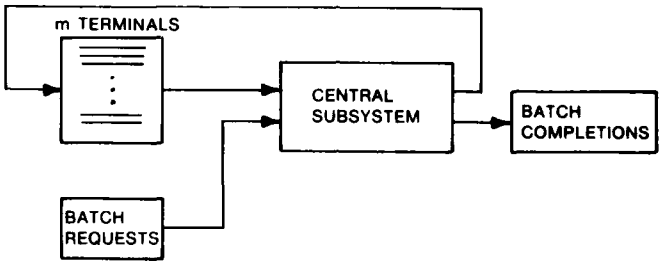


Figure 66. Interactive Computer System

The structural characteristics of the system are:

- o There is a Central Subsystem for processing jobs.
- o There are m interactive terminals in the system and each terminal is associated with a job.
- o Each terminal has a two-way communication path in which jobs flow to and from the Central Subsystem.
- o There is a Batch facility having two subsystems.
- o The Batch Requests subsystem has a one-way communication path in which jobs flow to the Central Subsystem.
- o The Batch Completions subsystem has a one-way communication path in which jobs are received from the Central Subsystem.

The functional characteristics of the system are:

- o Each terminal is manned by a user who alternates between thinking and waiting.
- o In the thinking state the user is contemplating what job to submit next to the Central Subsystem.

- o On submitting a job, the user enters the waiting state, where he remains until the central subsystem completes the job for him.
- o The Batch Requests subsystem comprises jobs submitted by other means, e.g. remote job entry stations. Batch jobs are submitted asynchronously, without regard to batch completions.
- o The Central Subsystem processes incoming jobs and returns them to the appropriate terminal or to the Batch completion subsystem. The central subsystem has a black box behavior at this level, which will be further described later.

The performance characteristics of the system are as:

- o Each terminal has a think time, Terminal_Think_Time, having a negative exponential distribution.
- o Batch requests are generated at the rate Batch_Request_Rate, having a negative exponential distribution. o Performance requirements on the central subsystem may include:
  - Maximum delay in processing jobs
  - Average delay in processing jobs

The next level of description of the Central Subsystem is shown in Figure 67. The structural characteristics are:

- o The Front End Processor (FEP) has communication paths to the terminals, the Batch subsystems, and the CPU.
- o The CPU is connected to the FEP and to n I/O Devices.
- o Each I/O Device has a communication path to/from the CPU.

The functional characteristics of the Central Subsystem are:

- o The FEP is the interface between the various incoming and outgoing jobs and the single CPU job stream.
- o The CPU has a queue in which arriving jobs enter, and from which jobs are selected to be processed (FIFO ordering is used). A processed job is completed, and is sent to the FEP, or requires I/O processing, and is sent to an I/O Device. The particular

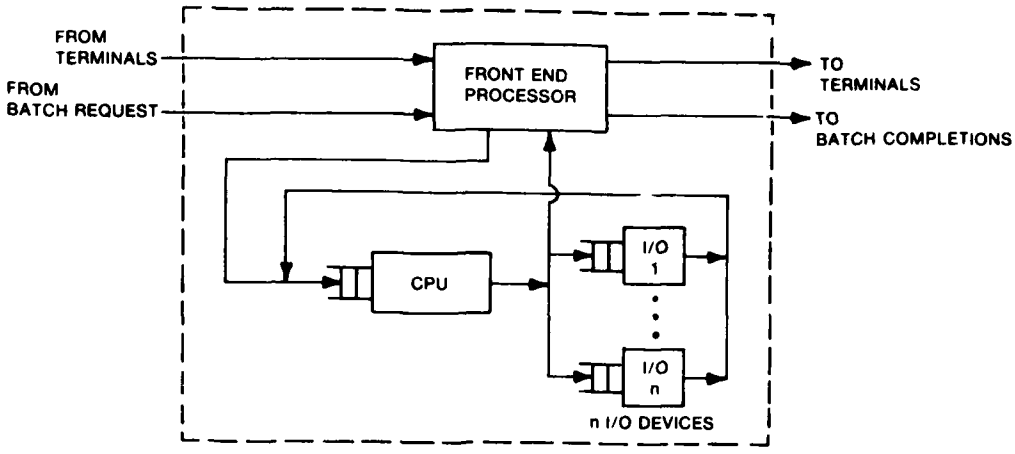


Figure 67. Central Subsystem

(Figure 67)

routing is described by a probability transition table, e.g., p(i) = probability of the job going to device i.

- o Each I/O Device has a queue in which incoming I/O job requests enter, and from which jobs to receive I/O processing are removed. Completed jobs are returned to the CPU.

The performance characteristics of the Central Subsystem are:

- o The FEP is assumed to cause no delay or overhead when handling jobs.
- o The CPU processes each job for an amount of time, CPU_Time. The distribution of the service time is negative exponential. Performance measures that can be taken include:
  - The size of the queue (both average and maximum size).
  - The utilization of the CPU.
  - The average waiting delay of jobs in the queue and the CPU.
  - The arrival rate of jobs to the CPU.
- o Each I/O Device can process an IO request in an amount of time, IO_Time, which also has a negative exponential distribution. Performance measures that can be taken are similar to those of the CPU.

The informal description of the multiprogrammed computer system is used to generate the formal CSDL definition. Figure 68 illustrates the high level structure of the interactive computer system using CSDL primitives. Abstract machines are used to represent:

- o Four terminals
- o Batch Requests and Batch Completion subsystems
- o Central Subsystem

CSDL channels are used to represent the communication topology. For example, TIC(1).out is an outlet at Terminal_Machine(1), and TIC(1).in is an inlet at Central_Subsystem_Machine, indicating a flow of information from the terminal to the Central Subsystem.

Figure 69 shows the CSDL structure of the Central_Subsystem_Machine. The CPU queue is represented by an abstract machine, CPU_Queue_Machine, in order to reduce the complexity of the CPU_Machine.

A skeleton CSDL description of the interactive computer system is included at the end of this section as subsection 6.6.

6.5 AN APPROACH TO PERFORMANCE ANALYSIS

Subsection 6.1 identifies four approaches to performance analysis. In review they are:

- o Algebraic manipulation
- o Queueing theory
- o Dynamic simulation
- o Observation of a realized system

The first approach provides results by generation and manipulation of static algebraic statements. The second approach provides a steady-state dynamic analysis by formal mathematics. The third approach provides a steady-state or transient dynamic analysis by logical experimentation, and the fourth approach provides transient or steady-state dynamic analyses by physical experimentation.

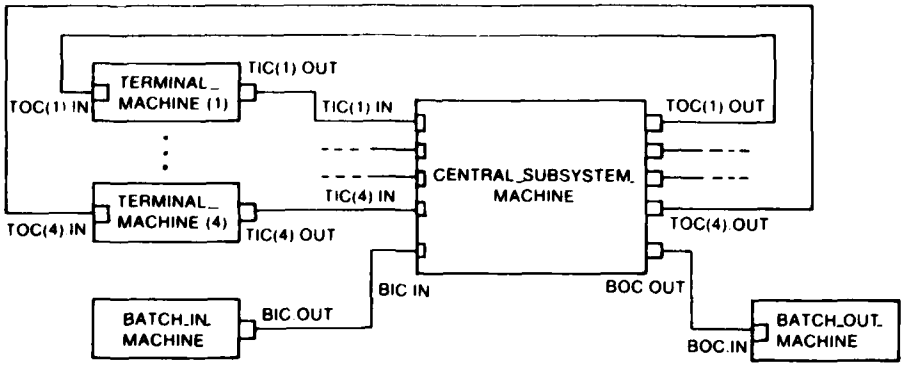


Figure 68. Structure of CSDL Definition of the Interactive Computer System

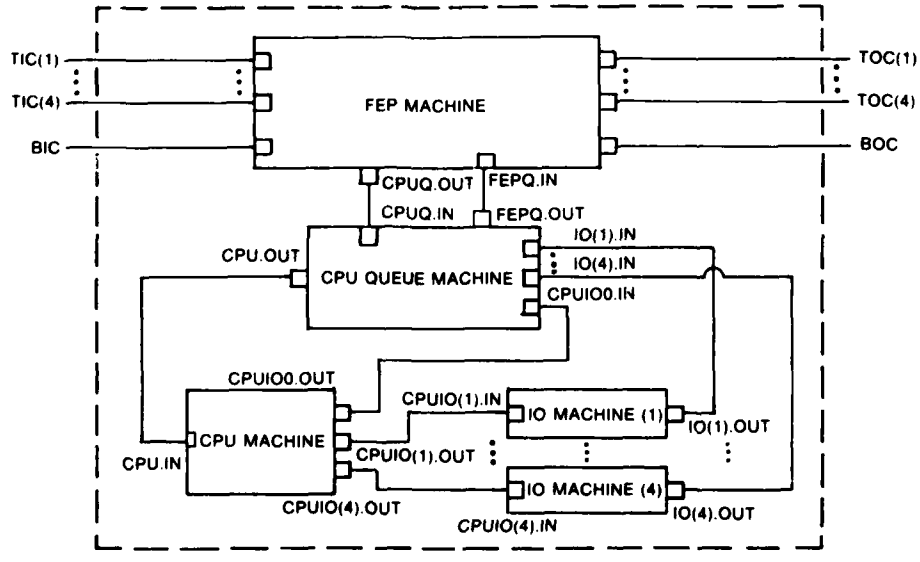


Figure 69. Structure of CSDL Definition of Central Subsystem

For all approaches the objective is to validate performance questments and answer performance questions using a CSDL system definition including performance statements as input. Conceptually, application of each approach proceeds as indicated next.

(1) Algebraic manipulation is carried out by algebraically combining and transforming statements (and perhaps questments) until expressions are obtained which answer (or contribute to answering) performance questions. The approach has been succesfully used [FRAN79], although it is tedious when manually applied. Computer supported application of the approach may be related to symbol manipulation [NGEW79] or symbolic execution [DARR78], [YAU81], the latter approach possessing a somewhat dynamic bent.

(2) Queueing theory, while applicable to all levels of system definition, is most meaningful when action and partitioned level system definitions exist. At these levels, network queueing theory [GELE80] becomes useful, with queueing nodes representing actions and channel transmissions and node-to-node transition probabilities reflecting the order in which the activities (events) represented by the nodes occur. While the mapping between actions, channels, etc., and a network queueing model may be straightforward, the approach may not produce useful results because of the limitations of network queueing theory such as limitations in the flow patterns, queueing disciplines and distributions accommodated. Nevertheless, some significant subset of CSDL definitions could be analyzed by network queueing theory, and the approach is attractive because of the extensive number of network queueing theory computer program packages which exist and which yield state probabilities and node utilization and queueing delay measures for each network node.

(3) Dynamic simulation (discrete event), like queueing theory, is applicable at all levels of system definition, but again like queueing theory, it is most applicable to action and partitioned levels definitions. At these level, actions, channels, types, and hence multimachine systems are represented by procedural descriptions that respond to and cause the events that describe the behavior of the system. If a process-oriented simulation approach is taken [FRAN77], then each action, type invocation procedure, and channel activity becomes associated

with a process, such that the set of processes interact and cooperate to mimic system behavior. Theoretically any reasonable performance measure can be estimated by dynamic stimulation, but the cost (in experiment time) may be large in some cases.

(4) Observation of operational behavior requires an operational system or system testbed [KAIN79] that can be configured to correspond to the CSDL system definition. This approach requires attention to more low-level system definition detail than the others, and thus, while viable, this approach is limiting.

Of the four approaches, queueing theory and simulation seem the most fertile to pursue. For both cases it is necessary to map CSDL system descriptions to a form suitable for performance analysis. Ideally statements and descriptions are used to drive an analysis that provides answers to questions and questments, all governed by a performance analysis tool that inputs the extended CSDL definition and automatically produces both a performance model and results relevant to questments and questions. Such a tool might be used in the manner indicated in Figure 70.

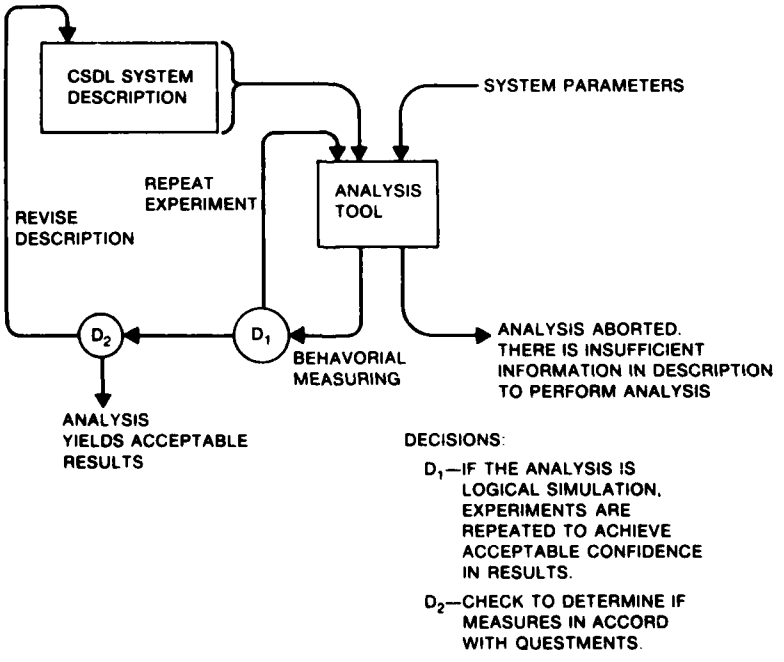


Figure 70. Performance Analysis During Design

Realization of such an ideal tool is as yet a remote prospect. It is not clear how close the functionality of a realizable tool can come to the ideal, nor is it clear what implications or restrictions the tool would have on CSDL and CSDL definitions. We do know that at certain stages of design, CSDL definitions will not contain enough information to conduct an analysis [LEVI79], and further that even when sufficient information is given, questions may only be answerable as derived measures of data obtainable from the analysis. Nevertheless pursuit of an interactive analysis tool employing queueing theoretic and discrete simulation seems to be a fruitful research arena.

6.6 A CSDL DEFINITION WITH PERFORMANCE SPECIFICATION

```
system ICS {Interactive Computer System}

machine Level1

   declarations
      types
      {Job is the entity, or message, that traverses the system. At this
       level, the form of a job is not known.}
      Job:abstract
      JobA:Job array
         let j:JobA
         invariant
            J.Dom=4
      end declarations

   partition
      interfaces
         {Specify two sets of channels of type Job}
         TIC,TOC:JobA;
         BIC,BOC:Job channel

      components
         {Associate the channels with the machines}
         T1:(TIC(1).out,TOC(1).in):Terminal_Machine(1);
         T2:(TIC(2).out,TOC(2).in):Terminal_Machine(2);
         T3:(TIC(3).out,TOC(3).in):Terminal_Machine(3);
         T4:(TIC(4).out,TOC(4).in):Terminal_Machine(4);

         B1:(BIC.out):Batch_In_Machine;
         B2:(BOC.in):Batch_Out_Machine;

         CSM:(TIC.in,TOC.out,BIC.in,BOC.out):Central_Subsystem_Machine
```

```
communication behavior
   function
      {In this section behavior specifications are used to indicate
       that jobs are initiated by each terminal machine, enter the
       Central_Subsystem, and return to the initiating terminal.
       Batch jobs are iniated by Batch_In_Machine, and Batch job
       completions go to Batch_Out_Machine.}

   perf
      {The requirement of the interactive user think time is:}
      ∀i,j:integer[clock(leaves(TIC(j).out)<i>) -
         clock(arrives(TOC(j).in)<i> =
            1 / dist(negexp,1/Terminal_Think_Time)] !? and

      {The requirement on the maximum allowable response time
       delay for jobs in the Central Subsystem is:}
      ∀i,j:integer[clock(leaves(TOC(i).out)<j>) -
         clock(arrives(TIC(i).in)<j>) ≤ Maximum_Response_Time] !? and

      {The above assertion can also be used to determine the measured
       response time for each job transaction:}
      ∀i,j:integer[clock(leaves(TOC(i).Out)<j>) -
         clock(arrives(TIC(i).in)<i>)] = ? and

      {The throughput rate for the i-th terminal is obtained by
       using a term defined in Section 6.3:}
      Outlet_Departure_Rate(TIC(i).out) = ? and

      {The rate of Batch requests can be specified by:}
      Outlet_Departure_Rate(BIC.out) = dist(negexp,Batch_Request_Rate) !?
         and

      {The maximum batch processing delay is:}
      ∀i:integer[clock(leaves(BIC.out)<i> -
         clock(arrives(BOC.in)<i>) ≤ Maximum_Batch_Delay] !? and

      {Specify that all communication connections (e.g. between
      each terminal and the Central_Subsystem) have zero time delay}
      ∀i,j:integer[clock(arrives(TIC(i).in<i>) -
         clock(leaves(TIC(i).out)<j>) = 0] ! and
      ∀i,j:integer[clock(arrives(TOC(i).in)<j> -
         clock(leaves(TOC(i).out)<j>) = 0] ! and
      ∀i:integer[clock(arrives(BIC.in)<i>) -
         clock(leaves(BIC.out)<i>) = 0] !

end partition

end Level1
```

```
machine Terminal_Machine(ID)
   {Generic specification of each terminal machine}

   declarations
      types
         Job:abstract
      objects
         TIC:Job Outlet;
         TOC:Job Inlet
      flows
         F1:Job from TOC to TIC

   end declarations

   specifications
      behavior
         function
         {Specify that completed jobs, which arrive from the
          Central Subsystem, are examined, and a new job request to
          the Central Subsystem is issued.}

         perf
         {At this level, the user think time appears as a specification
          that satisfies the performance requirement on the think time
          in the level1 machine.  Also, various compomponents of the think
          time could be specified.}
         ¥i:integer[clock(arrives(TOC<i>) - clock(leaves(TIC<i>) =
            1 / dist(negexp,1/Terminal_Think_Time)] !

   end specifications

end Terminal_Machine
```

```
machine Batch_In_Machine
   {Specification of the machine that generates batch requests.}

   declarations
      types
         rob:abstract
      objects
         BIC:Job outlet

   end declarations

   specifications
      behavior
         function
      {Specify the characteristics of the Batch request jobs.}

      perf
         {Specify the distribution of the times between Batch
          requests.}
         ∀i:integer[clock(leaves(BIC)<i>) - clock(leaves(BIC)<i>) =
            1 / dist(negexp,Batch_Request_Rate)] !

   end specifications

end Batch_In_Machine

machine Batch_Out_Machine
   {Specification of the machine that absorbs the batch requests.}

   declarations
      types
         Job:abstract
      objects
         BOC:Job inlet

   end declarations

   specifications
      behavior
         function
         {Specify any functional constraints on the completed Batch
          Requests.}
         perf
            {The performance requirements on batch processing delay that
             appeared in the levell machine now appear as specifications
             that satisfy those requirements.}

   end specifications

end Batch_Out_Machine
```

```
machine Central_Subsystem_Machine
   {Specification of the Central Subsystem}

   declarations
      types
         Job:abstract
      objects
         TIC(1),TIC(2),TIC(3),TIC(4),BIC:Job inlet;
         TOC(1),TOC(2),TOC(3),TOC(4),BOC:Job outlet

   end declarations

   specifications
      behavior
         function
            {Describe functional aspects of job processing.}

         perf
            {Performance requirements on the Central Subsystem were given
             in the Level1 machine definition. They can also appear here
             as requirements on the partitions of this machine.}

   end specifications

   partition
      {Define the internal structure of the Central Subsystem
       Machine, as shown in Figure 6-5.}

      interfaces
         {Specify the channels}
         CPUQ,FEPQ,CPU,CPUIOO:Job channel;
         CPUIO,IO;JobA channel

      components
         {Define the machines in the Central Subsystem.}
         FEP:(TIC.in,TOC.out,BIC.in,BOC.out,CPUQ.out,FEPQ.in):FEP_Machine;

         CQ:(CPUQ.in,FEPQ.out,IO.in,CPUIOO.in,CPU.out):CPU_Queue_Machine;

         CPU:(CPU.in,CPUIOO.out,CPUIO.out):CPU_Machine;

         IO1:(CPUIO(1).in,IO(1).out):IO_Machine(1);
         IO2:(CPUIO(2).in,IO(2).out):IO_Machine(2);
         IO3:(CPUIO(3).in,IO(3).out):IO_Machine(3);
         IO4:(CPUIO(4).in,IO(4).out):IO_Machine(4)
```

```
communication behavior
   function
      {Specify functional requirements on each of the machines in
       the Central Subsystem.}

   perf
      {Specify performance requirements on each of the machines
       in the Central Subsystem that might satisfy the performance
       requirements specified in the communication behavior section
       of Central_Subsystem_Machine.  These requirements are to
       be satisfied by the individual partititined machines.
       Their composite behavior must be ascertained by a system
       performance analysis.}

      {For correspondence with Central Server queueing networks
       of this type of computer system, we assume the FEP
       and CPU_Queue machines have zero delay time for jobs.}

      forall i,j:integer[
         clock(CPUQ.out)<i>) -
            clock(arrives(TIC(j)) or arrives(BIC)<i>) = 0] ! and
      forall i,j:integer[
         clock(leaves(TOC(j) or leaves(BOC))<i>) -
            clock(arrives(FEPQ.in)<i>) = 0] !  and

      {Specify the rate at which the CPU processes jobs (The queue
       for the CPU is in the CPU_Queue_Machine. The CPU_Machine is
       therefore a server for jobs.  Jobs leave to any of the
       outlets of the CPU_Machine.}
      ¥i,j:integer[
         clock(leaves(CPUIOO.out or leaves(CPUIO(j)<i>) -
            clock(arrives(CPU.in)<i>) =
               1 / dist(negexp,1/CPU_Time)] and

      {Specify the fraction of jobs proceeding next to each I/O
       Device, or back to the terminal after completing processing,
       using the set of routing probabilities p(i).}
      let n:integer such that [
         {n is the total umber of job arrivals}
         n = #i:integer[arrives(CPU.in)<i>] and
         {Probability of a completed job}
         #i:integer[leaves(CPUIOO.out)<i>] / n = dist(negexp,p(0)) ! and
         {Probability of a job being routed to I/O Device i}
         ¥i,j:integer[leaves(CPUIO(i).out)<j>] / n =
            dist(negexp,p(i))] ! and

         {Requirements on the delays at each of the IO devices}
         ¥i,j:integer[clock(Leaves(IO(j).out)<i>) -
            clock(arrives(CPUIO(j).in)<i> = 1/dist(negexp,1/IO_Time)] !?

   end partition
end Central_Subsystem_Machine
```

```
machine FEP_Machine

   declarations
      types
         Job:abstract
      objects
         TIC:JobA inlet;
         TOC:JobA outlet;
         BIC:Job inlet;
         BOC:Job outlet;
         CPUQ:Job outlet;
         FEPQ:Job inlet

   end declarations

   specifications
      behavior
         function
            {Specify that arriving jobs are queued and sent to
             CPU_Queue_Machine in First-In-First-Out order.
             Processed jobs are sent back to their appropriate
             Terminal, or to Batch Out.}

         perf
            {Specify that it is assumed that enqueueing and dequeueing is
             performed with zero time delay.}

   end specifications

end FEP_Machine
```

```
machine CPU_Queue_Machine

   declarations
      types
         Job:abstract
      objects
         CPUQ,CPUIOØ:Job inlet;
         IO:JobA inlet;
         FEPQ,CPU:Job outlet

   end declarations

   specifications
      behavior
         function
            {Specify that jobs arriving from the FEP_Machine and the I/O
             Devices are queued in FIFO order for the CPU.  Jobs arriving
             at the CPUIOØ inlet are completed, and are sent to the
             FEP_Machine.}

         perf
            {As stated earlier, it is assumed that overhead for queueing job
             involves zero time delay.  However, if we suppose the queue
             that is specified in the functional behavior is implemented
             by an array, then terms described in Section 6.3 can be used
             to refer to the average queue size, departure rate, and average
             waiting time of jobs.}
            FIFO_Average_Queue_Size(Queue) = ? and

            FIFO_Queue_Departure_Rate(Queue) = ? and

            Queue_Average_Waiting_Time(Queue) = ?

   end specifications

end CPU_Queue_Machine
```

```
machine CPU_Machine
   {Definition of the System's CPU processor.}

   declaration
      types
         Job:abstract
      objects
         CPU:Job inlet;
         CPUIOØ:Job outlet;
         CPUIO:JobA outlet

   end declarations

   specifications
      behavior
         function
            {Specify that a job is removed from the CPU inlet, is processed,
             sent (probabilistically according to higher level job routing
             specification) to one of the IO device outlets or to the job
             completion outlet (CPUIOO).  After a job is processed, another
             job is removed from the CPU inlet; all queueing is done in the
             CPU_Queue_Machine.}

         perf
            {Specify the mean service time of jobs.}
            ¥i,j:integer[clock(leaves(CPUIO(j) or CPUIOO)<i>) -
               clock(arrives(CPU)<i>) = 1/dist(negexp,1/CPU_Time)] ! and
            {Specify the CPU utilization - the fraction of time the CPU
             is busy}
            let s1:integer,z:integer array such that [
            {Determine each idle time between job arrivals at the CPU}
            ¥i,j:integer[z(i) = clock(arrives(CPU)<i+1>) -
               (clock(leaves(CPUIO(j)) or leaves(CPUIOO)<i>))] and
            {Add the idle times that are now in the z array}
            sum(s1,s,1) and

            {Determine utilization}
            1 - s1 / (clock(arrives(CPU)<#i[Arrives(CPU)]>) -
               clock(arrives(CPU)<1>) = ?

   end specifications

end CPU_Machine
```

```
machine IO_Machine(ID)
   {Generic Definition of the I/O Device Machines.}

   declarations
      types
         Job:abstract
      objects
         CPUIOLJob inlet;
         IO:Job outlet

   end declarations

   specifications
      behavior
         function
            {Specify that as jobs arrive they are queued in FIFO order,
             and depending on the kind of I/O Device being represented,
             there are various stages of IO processing, eg., seek,
             latency.  Introduce queue Q and action A to assist in the
             specification of job queueing and I/O processing.  Also,
             depending on the level of detail, various IO scheduling
             policies can be specified.}

         perf
            {The maximum delay for I/O processing was given in a higher
             level.  Here, the performance delay specification of each
             job I/O can be given, using action A.}

            let DP:real such that [
               DP = ∀i:integer[clock(last(A)<i>)-clock(first(A)<i>) =
                  1 / dist(negexp,1/IO_Time)]] ! and
            {Assuming the queue Q is implemented by an array QA, we
             can specify the average waiting time of jobs in the queue.}
            let DQ:real such that [
               DQ = Queue_Average_Waiting_Time(QA)] ! and
            {The queueing delay plus the I/O processing delay is the I/O
             device delay.}
            ∀i:integer[clock(leaves(IO)<i>) - clock(arrives(CPUIO)<i>)] =
               PD + DQ !
            {This delay must now be compared with the requirements, given
             at a higher level of detail, on the performance of the I/o
             Devices.}

   end specifications

end IO_Machine

end ICS
```

BIBLIOGRAPHY

[AMBL77] A. L. Ambler, "Gypsy: A Language for Specification and Implementation of Verifiable Programs," ACM SIGPLAN Notices, vol. 12, no. 3, Mar. 1977.

[BELA79] L. A. Belady and M. M. Lehman, "The Characteristics of Software Systems," in Research Directions in Software Technology, P. Wegner ed., MIT Press, 1979.

[BELA80] L. A. Belady and B. Leavenworth, "Program Modifiability," in Software Engineering, H. Freeman and P. M. Lewis II, eds., Academic Press, 1980.

[BELL77] T. E. Bell, D. C. Bixler, M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Software Engineering, vol. SE-3, no. 1, Jan. 1977.

[BERG80] H. K. Berg, "DST - A Distributed System Testbed," Proc. Honeywell International Conf. on Database Management Systems, Oct. 1980.

[BERG81] H. K. Berg and W. T. Wood, "The Impact of Distributed Processing on Software Design," Tech. Report HR81-256:17-38, Honeywell Corporate Computer Sciences Center, 1981.

[BOCH79] G. V. Bochmann, "Architecture of Distributed Computer Systems," Lecture Notes in Computer Science, vol. 77, Springer-Verlag, 1979.

[BOEB78] W. E. Boebert, W. R. Franta, E. D. Jensen and R. Y. Kain, "Kernel Primitives of the HXDP Executive," COMPSAC 78, Nov. 1978.

[BOEH79] B. W. Boehm, "Software Engineering As It Is," Proc. 4th International Conference on Software Engineering, Munich, 1979.

[BOOT79] T. L. Booth, "Performance Optimization of Software Systems Processing Information Sequences Modeled by Probabilistic Languages," IEEE Trans. on Software Engineering, vol. SE-5, no. 1, Jan. 1979.

[BOOT80] T. L. Booth and C. A. Wiecek, "Performance of Abstract Data Types as a Tool in Software Performance Analysis and Design," IEEE Trans. on Software Engineering, vol. SE-6, no. 2, Mar. 1980.

[BOYD78a] D. L. Boyd, A. Pizzarello and S. C. Vestal, The Rational Design Methodology - Final Report, RADC Contract No. F30602-77-C0043, Honeywell Inc., June 1978.

[BOYD78b] D. L. Boyd and A. Pizzarello, "Introduction to the WELLMADE Design Methodology," IEEE Trans. on Software Engineering, vol. SE-4, no. 4, July 1978.

[BRIN73] P. Brinch Hanson, Operating System Principles, Prentice-Hall, 1973.

[BUZE76] J. P. Buzen, "Fundamental Operational Laws of Computer System Performance," ACTA Informatica, vol. 7, 1976.

[CAVO81] J. C. Cavouras and R. H. Davis, "Simulation Tools in Computer System Design Methodologies," The Computer Journal, vol. 24, no. 1, Jan. 1981.

[CHAN79] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," IEEE Trans. on Software Engineering, vol. SE-5, no. 5, Sep. 1979.

[COOK80] R. P. Cook, "The Starmod Distributed Programming System," Proc. COMPCON 80 Fall, Sep. 1980.

[DARR78] J. A. Darringer and J. C. King, "Application of Symbolic Execution to Program Testing," Computer, vol. 11, no. 4, 1978.

[DEBA75] J. W. DeBakker, "The Fixed Point Approach in Semantics: Theory and Applicaitons," in Foundations of Computer Science, J. W. DeBakker, ed., Mathematical Centre Tracts 63, Amsterdam, 1975.

[DENN78] P. J. Denning and J. P. Buzen, "The Operational Analysis of Queueing Network Models," Computing Surveys, vol. 10, no. 3, Sept. 1978.

[DENN80] P. J. Denning, "What is Experimental Computer Science?," Comm. of the ACM, vol. 23, no. 10, Oct. 1980.

[DIJK76] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.

[ENDE72] H. B. Enderton, A Mathematical Introduction to Logic, Academic Press, New York, 1972.

[FELD79] J. A. Feldman, "High Level Programming for Distributed Computing," Comm. of the ACM, vol. 22, no. 6, June 1979.

[FORE79] I. Foreman, "Data Abstraction in the Design of SWISP," Presented at 19th Annual NBS/ACM Technical Symposium, June 1979.

[FRAN77] W. R. Franta, The Process View of Simulation, Elsevier North-Holland, 1977.

[FRAN79] W. R. Franta, W. E. Boebert and H. K. Berg, "An Approach to the Specification of Distributed Software," in The Use of Formal Specification of Software, H. K. Berg and W. K. Giloi, eds., Springer-Verlag, 1979.

[GELE80] E. Gelenbe and I. Mitrani, Analysis and Synthesis of Computer Systems, Academic Press, 1980.

[GERH76] S. L. Gerhart, L. Yelowitz, "Observations on the Fallibility in Application of Modern Programming Methodologies," IEEE Trans. on Software Engineering, vol. SE-2, no. 3, Sept. 1976.

[HOAR78] C. A. R. Hoare, "Communicating Sequential Processes," Comm. of the ACM, vol. 21, no. 8, Aug. 1978.

[HONE80] Honeywell Inc., Reference Manual for the Ada Programming Language, United States Department of Defense, 1980.

[ISO80] International Standards Organization, "Open Systems Interconnection Reference Model," ISO/TC97/SC16N, May 1980.

[KAIN79] R. Y. Kain, W. R. Franta and G. D. Jelatis, "CHIMPNET: A Network Testbed," Computer Networks, vol. 3, no. 6, Dec. 1979.

[LEVI79] K. N. Levitt, "A Basis for Simulating Modules Written in Special," SRI Project 4828 report on Contract N00123-76-C-0195, SRI International, Aug. 1979.

[LISK77] B. H. Liskov and V. Berzins, "An Appraisal of Program Specifications," Computation Structures Group Memo 141-1, MIT Laboratory of Computer Science, Apr. 1977.

[MARI80] M. P. Mariani and D. F. Palmer, "Distributed System Design," Tutorial Notes, COMPCON 80 Fall, Sept. 1980.

[NELS78] R. Nelson, "Software Data Collection and Analysis," Draft Partial Report, RADC, Rome, New York, 1978.

[NG79] E. W. Ng, "Symbolic and Algebraic Computation," Lecture Notes in Computer Science, vol. 72, Springer-Verlag, 1979.

[RAMA80] C. V. Ramamoorthy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," IEEE Trans. on Software Engineering, vol. SE-6, no. 5, Sept. 1980.

[REED79] D. P. Reed and R. K. Kanodia, "Synchronization with Eventcounts and Sequencers," Comm. of ACM, vol. 22, no. 2, Feb. 1979.

[SANG79] J. Sanguinetti, "A Technique for Integrating Simulation and System Design," Conference on Simulation, Measurement, and Modeling of Computer Systems, 1979.

[SAUE79] C. H. Sauer and E. A. MacNair, "Queueing Network Software for Systems Modeling," Software Programming Experience, vol. 9, 1979.

[SILB81] A. Silberschatz, "Port Directed Communication," The Computer Journal, vol. 24, no. 1, The British Computer Society, Feb. 1981.

[TEIC77] D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, vol. SE-3, no. 1, Jan. 1977.

[VERA79] M. Veran, "QNAP Description Language," Research Report, Antenne Scientifique Cii Honeywell-Bull, Grenoble, France, 1979.

[WANG81] P. S. Wang, "DST User's Guide," Tech. Report, Honeywell Corporate Computer Sciences Center, 1981.

[WEGB76] B. Wegbreit, "Verifying Program Performance," JACM, vol. 23, no. 4, Oct. 1976.

[WIRT77] N. Wirth, "Molula: A Language for Modular Multiprogramming," Software Practice and Experience, vol. 7, 1977. pp. 3-84.

[YAU81] S. S. Yau, C. C. Yang, and S. M. Shatz, "An Approach to Distributed Computing System Software Design," IEEE Trans. on Software Engineering, vol. SE-7, no. 4, July, 1981.

APPENDIX A
CSDL SUMMARY

An informal description of the syntax and semantics of the concurrent system description language (CSDL) is presented in this appendix. CSDL includes notation for design specifications, design descriptions, and document organization. A formal specification of the syntax is presented using Backus-Naur notation in Appendix B, and as syntax diagrams in Appendix C.

NOTATION CONVENTIONS

[ ] information between brackets is optional

< > denotes a syntactic symbol

{ } denotes explanatory information

DOCUMENT ORGANIZATION

Comments can occur anywhere as {<text>}

1. System Definition Text:

```
system <system id>

   <machine id>
   <machine id>
   .
   .
   .

<machine definition text>
<machine definition text>
.
.
.

end <system id>
```

2. Machine Definition Text:
{a machine definition text may contain either
the definition of one component or the
refinement of one or more related types}

machine <machine id>

[refines <type id>]

<declarations chapter>

<specifications chapter>

<partition chapter>
{only exists for partitioned machines}

<programs chapter>
{only exists for simple machines}

[refines <type id>
.
.
.

refines <type id>
.
.
.]

end <machine id>

3. Declarations Chapter:

declarations

[predicates
<predicate declarations>]

[types
<type declarations>]

objects
<object declarations>

[actions
<action declarations>]

[flows
<flow declarations>]

end declarations

4. Specifications Chapter:

```
specifications

   [mappings
      {only exists for type refinement machines}
      <mapping specifications>]

   [states
      [init <assertions>]
      [final <assertions>]
      [invariant <assertions>]]

   [behavior
      [function <temporal assertions>]
      [perf <performance assertions>]]

end specifications
```

5. Partition Chapter:
   {only exists for partitioned machines}

```
partition

   interfaces
      <interface declarations>

   [paths
      <path declarations>]

   components
      <component declarations>

   [communication behavior
      [function <temporal assertions>]
      [perf <performance assertions>]]

end partition
```

6. Programs Chapter:
{only exists for simple machines}

programs

<program definition text>
<program definition text>
.
.
.

end programs

7. Program Definition Text

program <program id>[(<parameters>)]
[returns <parameter>]

pre
<assertions>

post
<assertions>

[invariant
<assertions>]

[perf
<performance assertions>]

[[variables
<variable declarations>]

text
<program text>]

end <program id>

DECLARATIONS

1. Predicates

```
<predicate id>[(<parameters>)] means <assertion>
<predicate id>[(<parameters>)] characterized by <assertion>

e.g.  INDEX(A:char array,i:integer) means
           A.lob ≤ i ≤ A.hib
```

2. Types

```
<type id>[(parameters>)]:<type expression>
     [[let <object declarations>]
     [init <assertions>]
     [invariant <assertions>]
     [ofun <function id>[(<parameters>)]
          [returns <parameters>]
          pre <assertions>
          post <assertions>
          [perf <performance assertion>]]
     [vfun <function id>[(<parameters>)]
          returns <parameter>
          pre <assertions>
          post <assertions>
          [perf <performance assertion>]]
     end <type id>]

e.g.

stack(n:integer):name array
     let n:integer, S:stack(n)
     init S.dom=0
     invariant S.dom≤n
     ofun push (X:name) returns S
          pre S.dom<n
          post S'=S.hiext(X)
     vfun full returns b:boolean
          pre true
          post [S.dom=n => b=true] and
               [S.dom<n => b=false]
     end stack
```

3. Objects (Variables)

<object id>:<type expression>
<object id>:<type id>[(<parameters>)]

e.g. X:integer
Y:stack(100)

4. Actions

<action id> [(<parameters>)] means
pre <assertion>
post <assertion>
perf <performance assertion>

e.g.

empty (buffer: char array) means
pre buffer.dom ≥ 0
post buffer'.dom = 0

5. Flows

<flow id>:<type id> from <object ref> to <object ref>

e.g. F1:integer from A to B

6. Interfaces

<object id>:<type expression>

e.g. C:integer channel

7. Paths

<flow id> passes through <object ref list>

e.g. F1 passes through C

8. Components

<component id> (<object ref list>):<machine id>[(<parameters>)]

e.g. comp1(A,C.out):machine_a

SPECIFICATIONS

1. Mappings:

let <object declarations>
[<rep clause>;]
[if <rep clause> then <rep clause>;]
[if <assertion> then <rep clause>;]

where <rep clause> is:

<value ref> represents <value ref>
<assertion> represents <predicate id>[(<parameters)]
<function ref> represents <function ref>
<program ref> represents <function ref>

2. Assertions:

a. Primitives

Arithmetic expression on:

object references
value constants
bracketed expressions
function references
program references

b. Relational connectors

<, ≤, >, ≥, =, ≠

c. Quantifiers

exististential - ∃, for some, ∃!, for some unique
universal - ∀, for all

d. Logical connectors

or, and, xor, not,
if ... then, =>, iff, <=>

Examples

for some i:integer [pl(i).name=id]

∀ i:integer [p≤i≤pg.hib =>
pg'(i+1)=pg(i) & ps'(i-1)=pg(i)]

3. Temporal Assertions

a. Primitives

(1) gets(<interface object ref>)
arrives(<interface object ref>)
puts(<interface object ref>)
leaves(<interface object ref>)
<value ref>

(2) <value ref> prior <event><<event ordinal>>
<value ref> after <event><<event ordinal>>

where <value ref> is one of the primitives listed under Assertions

b. Events

occurs(<assertion>)<<event ordinal>>
<event fun>(<function ref>)<<event ordinal>>
<event fun>(<program ref>)<<event ordinal>>
<event fun>(<action ref>)<<event ordinal>>
<event fun>(<assertion>)<<event ordinal>>

where <event fun> is effects, first, or last;
<assertion> is assertion on primitives in category 1

c. Relational connectors (between primitives in category 2)

<, ≤, >, ≥, =, ≠

d. Temporal connectors (between events)

precedes, before, later

e. Quantifiers

existential - ∃, for some, ∃!, for some unique
universal - ∀, for all

e. Logical connectors

or, and, xor, not,
if ... then ,=>, iff, <=>

Examples

∀ i:posint [occurs(line.dom=0)<i> later
effects (program1)<i>]

forall i:posint
[effects(arrives (inlet1) or arrives(inlet2))<i>
later leaves(outleta)<i>]

4. Performance Assertions

a. Primitives

Arithmetic expression on:

clock(<event>)
dist(<arith expression>)

where <arith expression> is on primitives listed under Assertions

b. Relational connectors

<, ≤, >, ≥, =, ≠

c. Quantifiers

exististential - ⊣, for some, ⊣!, for some unique
universal - ∀, for all

d. Logical connectors

or, and, xor, not,
if ... then, =>, iff, <=>

e. Terminators

! - statements
? - questions
!? - questments

DATA TYPES

{t is a type expression}

1. Primitive Data Types
   and their operations

   a. integer

      {unary} -, +
      {binary} +, -, *, ÷, =, ≠, <, ≤, >, ≥
      {assignment} :=

   b. real

      {unary} -, +
      {binary} +, -, *, ÷, =, ≠, <, ≤, >, ≥
      {assignment} :=

   c. boolean

      {constants} T, F
      {unary} not
      {binary} and, or, xor, =, ≠
      {conditional} cand, cor
                    (valid only in guard expressions)
      {assignment} :=

   d. char

      {binary} =, ≠
      {assignment} :=

   e. (V1[]V2[]...[]Vn) {enumerated}

      where V1, V2, ..., Vn are the only permissible values
      of this data type

      e.g. COLOR: (RED[]WHITE[]BLUE)

   f. abstract

      {a type whose definition is given elsewhere}

2. Structured Data Types
and their operations

a. Cartesian Product {Record}

(O1:t1, O2:t2, ..., On:tn)

selector operation:

if A is a cartesian product then
A.Oi selects the ith component

b. Discriminated Union

(V1:t1 [] V2:t2 [] ... [] Vn:tn)

tag operation:

if A is a discriminated union then A.tag has the value
Vi if the value of A is of type ti

c. Array

t array

operations:

if A is an array of items of type t then

A.lob -	the smallest index value
A.hib -	the largest index value
A.dom -	the number of elements; A.dom = A.hib-A.lob+1 $\geq$ 0
A(I) -	the value of the I-th item
A.low -	the first item
A.high -	the last item
A.lorem -	remove the first item
A.hirem -	remove the last item
A.loext(a) -	append a new first item, a
A.hiext(a) -	append a new last item, a
A:=(k,a1,...,an) -	initialize A with A.lob=k, and A(k)=a1, ..., A(k+n-1)=an
A.swap(I,J) -	exchange the Ith and Jth item

3. Interface Data Types
and their operations

a. Outlet

t outlet:(flag:boolean,window:t)

operations:

if A is an outlet of type t then

A.put(a) - put value a in A.window
set A.flag to F

A.went - check the value of A.flag
{A.flag will be T when it is safe to put a new value in A.window without overlaying the previous value}

b. Inlet

t inlet:(flag:boolean,window:t)

operations:

if A is an inlet of type t then

A.get - get value from A.window
set A.flag to F

A.came - check the value of A.flag
{A.flag will be T when a new value is available in A.window}

c. Channel

t channel:(in:t inlet, out:t outlet)

operations:

{channel operations are operations on its inlet and outlet}

if A is a channel of type t then the following are valid operations on A:

A.in.get
A.in.came
A.out.put
A.out.went

PROGRAM STATEMENTS {P-NOTATION}

1. Sequential operations

    S1;S2 {Si - statement}

2. Assignment

    X1, X2, ..., Xn := E1, E2, ..., En

    The value of the expression Ei is assigned to be the value of the variable Xi. All Ei's are first evaluated, then assignments are made to the corresponding Xi's.

3. Selection

    if B2->S1[]B2->S2[]...[]Bn->Sn fi

    where B1, B2, ...,Bn are boolean-valued guard expressions, and S1, S2, ..., Sn are program statements.

    A true guard Bi is selected and the corresponding guarded command Si is executed. At least one guard must be true, else the program aborts.

4. Iteration

    do B1->S1[]B2->S2[]...[]Bn->Sn od

    where B1,B2, ..., Bn are boolean-valued guard expressions and S1, S2, ..., Sn are program statements. A true guard B1 is selected, the corresponding guarded command is executed, and the process is repeated. When all guards are false, a skip occurs.

5. No operation

    Skip

7. Functions

    Type operations

    e.g. A.hiext(I), B.get, C+D*E

    Programs

    <program id>(<actual parameters>)

APPENDIX B

CDSL SYNTAX - BNF

This appendix presents a formal specification of the CSDL syntax in Backus-Naur notation. Appendix C presents the same syntax in diagram form. The line numbers are referred to in Sections B.8 and B.9, which contains the cross references of the nonterminals and terminals, respectively. A nonterminal preceded by an * denotes the beginning of a sequence of production rules corresponding to the diagram defining the same nonterminal in Appendix C.

B.1 Document Organization

```
 1 *<CSDL text> ::= system <system id>
 2     <machine id list> <machine definition list> end  <system id>

 3  <machine id list> ::= <machine id> | <machine id>
 4     <machine id list>

 5  <machine definition list> ::= <machine definition text> |
 6     <machine definition text> <machine definition list>

 7 *<machine definition text> ::= machine <machine id> <parameters>
 8     <machine documentation text> end <machine id>

 9  <machine documentation text> ::= <refinement list> |
10     <design documentation text>

11  <refinement list> ::= <refinement> |
12     <refinement> <refinement list>

13  <refinement> ::= refines  <type id> <design documentation text>

14 *<design documentation text> ::= <declarations chapter>
15     <specifications chapter> <partition chapter> |
16     <declarations chapter> <specifications chapter>
17     <programs chapter>

18 *<declarations chapter> ::= declarations <predicates section>
19     <types section> <objects section> <actions section>
20     <flows section> end declarations

21 *<specifications chapter> ::= specifications <mappings section>
22     <states section> <behavior section> end specifications

23 *<partition chapter> ::= partition <interfaces section>
24     <paths section> <components section>
25     <communication behavior section> end partition
```

```
26 *<programs chapter> ::= programs <program definition list> end
27     programs

28  <program definition list> ::= <program definition section> |
29     <program definition section> <program definition list>

30 *<system id> ::= <identifier>

31 *<machine id> ::= <identifier>
```

B.2 Declarations Chapter

```
32 *<predicates section> ::= <nil> | predicates <predicate list>

33  <predicate list> ::= <predicate def> |
34     <predicate def> ; <predicate list>

35  <predicate def> ::= <predicate id> ( <formal parameter list> )
36     means <assertion> | <predicate id>
37     ( <formal parameter list> ) characterized by <assertion> |
38     <predicate id> ( <formal parameter list> ) means abstract |
39     <predicate id> means abstract | <predicate id> means <assertion> |
40     <predicate id> characterized by <assertion>

41  <formal parameter list> ::= <formal parameter> |
42     <formal parameter> , <formal parameter list>

43 *<formal parameter> ::= obj <formal object> | <formal value>

44 *<formal value> ::= <value id list> : <type expression>

45  <value id list> ::= <value id> | <value id> , <value id list>

46 *<types section> ::= <nil> | types <type list>

47  <type list> ::= <type> | <type> ; <type list>

48  <type> ::= <type id> : abstract |
49     <type id> ( <type parameter list> ) : <type expression>
50     <type specification> |
51     <type id> : <type expression> <type specification>

52  <type parameter list> ::=
53    <type parameter> , <type parameter list>

54  <type parameter> ::= <value id> : typename | <formal value>
55 *<type expression> ::= <standard type> | <structured type>
```

```
56  <standard type> ::= real | integer | boolean | character |
57     ( <enumeration> )

58  <structured type> ::= ( <cartesian product> ) | ( <union> ) |
59     <array type> | <interface type>

60  <enumeration> ::= <value id> | <value id> [] <enumeration>

61  <cartesian product> ::= <structure head> |
62     <structure head> , <cartesian product>

63  <union> ::= <structure head> | <structure head> [] <union>

64  <array type> ::= <type expression> array

65  <interface type> ::= <type expression> inlet |
66     <type expression> outlet | <type expression> channel

67 *<structure head> ::= <element id> : <type expression>

68 *<type specification> ::= <nil> | let <formal parameter list>
69     <init spec> <invariant spec> <function list> end <type id>

70  <init spec> ::= <nil> | init <assertion>

71  <invariant spec> ::= <nil> | invariant <assertion>

72  <function list> ::= <nil> | <function> |
73     <function> <function list>

74  <function> ::= ofun <function def> | vfun <function def>

75 *<function def> ::= <function id> <parameters> <output parameter>
76     <static spec> <performance spec>

77  <static spec> ::= pre <assertion> post <assertion>

78 *<parameters> ::= <nil> | ( <formal value list> )

79 *<output parameter> ::= <nil> | returns <value id> : <type id>

80  <formal value list> ::= <formal value> |
81     <formal value> , <formal value list>

82 *<performance spec> ::= <nil> | perf <performance assertion>
```

```
 83 *<objects section> ::= objects <object list>

 84  <object list> ::= <object> | <object> ; <object list>

 85  <object> ::= <object id list> : <object decl>

 86  <object id list> ::= <object id> | <object id> , <object id list>

 87  <object decl> ::= <type id> | <type expression> |
 88     <type id> ( <object parameter list> )

 89  <object parameter list> ::= <object parameter> |
 90     <object parameter> , <object parameter list>

 91  <object parameter> ::= <type id> | <arith expression> |
 92     <bool expression>

 93 *<actions section> ::= <nil> | actions <action list>

 94  <action list> ::= <action decl> | <action decl> ; <action list>

 95  <action decl> ::= <action id> ( <formal parameter list> ) means
 96     <action def> | <action id> means <action def>

 97  <action def> ::= abstract | <static spec> <performance spec>

 98 *<flows section> ::= <nil> | flows <flow list>

 99  <flow list> ::= <flow def> | <flow def> ; <flow list>

100  <flow def> ::= <flow id> : <type id> from <object ref> to
101     <object ref>

102 *<predicate id> ::= <identifier>

103 *<value id> ::= <identifier>

104 *<type id> ::= <identifier>

105 *<element id> ::= <identifier>

106 *<function id> ::= <identifier>

107 *<object id> ::= <identifier>

108 *<action id> ::= <identifier>

109 *<flow id> ::= <identifier>
```

B.3 Specifications Chapter

```
110 *<mappings section> ::= <nil> | mappings <representation list>

111  <representation list> ::= <representation> |
112     <representation> ; <representation list>

113  <representation> ::= let <formal parameter list> <rep term list>

114  <rep term list> ::= <rep term> | <rep term> ; <rep term list>

115  <rep term> ::=
116     if <representation clause> then <representation clause> |
117     if <predicate expression> then <representation clause>

118 *<representation clause> ::= <value ref> represents <value ref> |
119     <predicate expression> represents <predicate id> <parameters>

120 *<states section> ::= states <init spec> <final spec>
121     <invariant spec>

122  <final spec> ::= <nil> | final <assertion>

123 *<behavior section> ::= behavior <performance spec> |
124     behavior function <assertion> <performance spec>
```

B.4 Partition Chapter

```
125 *<interfaces section> ::= interfaces <channel list>

126  <channel list> ::= <channel def> | <channel def> ; <channel list>

127  <channel def> ::= <object id list> : <type expression>

128 *<paths section> ::= paths <passes through list>

129  <passes through list> ::= <passes through def> |
130     <passes through def> ; <passes through list>

131  <passes through def> ::= <flow id> passes through
132     <object ref list>

133  <object ref list> ::= <object ref> | <object ref> ,
134     <object ref list>
```

```
135 *<components section> ::= components <component list>

136  <component list> ::= <component def> |
137     <component def> ; <component list>

138  <component def> ::= <component id> ( <object ref list> ) :
139     <machine id> | <component id> ( <object ref list> ) :
140     <machine id> ( <machine parameter list> )

141  <machine parameter list> ::= <machine parameter> |
142     <machine parameter> , <machine parameter list>

143  <machine parameter> ::= <arith expression> | <bool expression>

144 *<communication behavior section> ::= communication behavior
145     <performance spec> | function <assertion> <performance spec>
```

B.5 Programs Chapter

```
146 *<program definition section> ::= <nil> | <program header>
147     <program spec> <program desc> end <program id>

148  <program header> ::=
149     program <program id> ( <formal parameter list> ) <output parameter> |
150     program <program id> <output parameter>

151 *<program spec> ::= <static spec> <performance spec>
152     <invariant spec>

153 *<program desc> ::= <nil> | variables <formal object list> text
154     <p notation>

155  <formal object> ::= <object id list> : <type expression>

156  <formal object list> ::= <formal object> |
157     <formal object> ; <formal object list>

158 *<program id> ::= <identifier>

159 *<component id> ::= <identifier>
```

B.6 Assertions

```
160 *<assertion> ::= <nil> | <predicate expression> |
161     <predicate expression> ; <assertion>

162 *<performance assertion> ::= <performance expression> |
163     <performance expression> ; <performance assertion>

164  <performance expression> ::= <predicate expression> |
165     <predicate expression> ! | <predicate expression> ? |
166     <predicate expression> !?

167 *<predicate expression> ::= <afactor> | <afactor> <implying op>
168     <afactor> | if <afactor> then <afactor>

169 *<afactor> ::= <clause> | <clause> <combining op> <afactor>

170  <implying op> ::= => | iff | '<=>

171  <clause> ::= <predicate> | <negating op> <predicate>

172  <combining op> ::= and | & | xor | or

173  <negating op> ::= not | ~

174 *<predicate> ::= <boolean literal> | <postfix pred> |
175     <relation chain> | <event predicate> |
176     [ <predicate expression> ] |
177     <quantifier clause> [ <predicate expression> ]
178     <let clause> [ <predicate expression> ] |

179 *<boolean literal> ::= true | false

180 *<postfix pred> ::= <predicate id> |
181     <predicate id> ( <actual parameter list> ) |
182     gets ( <object ref> ) | arrives ( <object ref> ) |
183     puts (<object ref> ) | leaves ( <object ref> )

184  <actual parameter list> ::= <actual parameter> |
185     <actual parameter> , <actual parameter list>

186 *<relation chain> ::= <aterm list> <relation tail>

187  <aterm list> ::= <aterm> | <aterm> , <aterm list>

188  <relation tail> ::= <relational op> <aterm list> |
189     <relational op> <aterm list> <relation tail>

190  <relational op> ::= '> | '< | = | ≠ | '≤ | '≥
```

```
191 *<event predicate> ::= <event term> <event predicate tail>

192  <event predicate tail> ::= <ordering op> <event term> |
193     <ordering op> <event term> <event predicate tail>

194  <ordering op> ::= before | precedes | later

195 *<quantifier clause> ::= <quantifier> <formal parameter list> |
196     <quantifier> <formal parameter list> <quantifier clause>

197  <quantifier> ::= ∃ | ∃! | for some | for some unique | ∀ |
198     for all

199 *<let clause> ::= let <let clause tail>

200  <let clause tail> ::= <formal parameter list> |
201     <formal parameter list> such that <predicate expression> |
202     <formal parameter list> and <let clause tail> |
203     <formal parameter list> such that <predicate expression> and
204     <let clause tail>

205 *<aterm> ::= <arith expression> | <temporal term> | <cardinality>

206 *<temporal term> ::=
207     <value ref> prior <event term> |
208     <value ref> after <event term>

209  <cardinality> ::= # <formal parameter list> [ <predicate expression> ]

210 *<event term> ::= <event primary> '< <arith expression> '>

211 *<event primary> ::= occurs ( <predicate expression> ) |
212     effects ( <action ref> ) | effects ( <predicate expression> ) |
213     effects ( <program activation> ) | first ( <action ref> ) |
214     first ( <predicate expression> ) | first ( <program activation> ) |
215     last ( <action ref> ) | last ( <predicate expression> ) |
216     last ( <program activation> )

217 *<action ref> ::= <action id> ( <actual parameter list> )
```

B.7 P Notation

```
218 *<p notation> ::= <statement> | <statement> ; <p notation>

219  <statement> ::= <simple statement> | <control statement>

220  <simple statement> ::= skip | abort | <assignment statement>
221     <program activation>

222  <control statement> ::= <alternative construct> | <repetitive construct>

223  <assignment statement> ::= <object ref list> := <value ref list>

224  <alternative construct> ::= if <guarded command set> fi

225  <repetitive construct> ::= do <guarded command set> od

226  <value ref list> ::= <value ref> | <value ref> , <value ref list>

227 *<program activation> ::= <program id> |
228     <program id> ( <actual parameter list> ) |
229     <object ref> . <function id> |
230     <object ref> . <function id> ( <value ref list> )

231 *<actual parameter> ::= <object ref> | <value ref>

232 *<object ref> ::= <object ref head> |
233     <object ref head> <object ref tail>

234  <object ref head> ::= <object id> | <value id> | <object id> '  |
235     <value id> '

236  <element id string> ::= . <element id> | ( <arith expression> ) |
237     . <element id> <object ref tail> |
238     ( <arith expression> ) <object ref tail>

239 *<value ref> ::= <arith expression> | <bool expression> |
240     <program activation>

241 *<arith expression> ::= <arith term> | <arith term> <adding op>
242     <arith expression>

243  <adding op> ::= + | -

244 *<arith term> ::= <arith factor> | <arith factor> <multiplying op>
245     <arith term>

246  <multiplying op> ::= * | / |  mod | * | div
```

```
247 *<arith factor> ::= <arith primary> | <unary op> <arith primary>

248  <unary op> ::= + | -

249 *<arith primary> ::= <object ref> | <constant> |
250     <program activation> | ( <arith expression> ) |
251     clock ( <event term> ) | dist ( <arith expression> )

252 *<guarded command set> ::= <guarded command> |
253     <guarded command> [] <guarded command set>

254  <guarded command> ::= <guard> --> <p notation>

255 *<guard> ::= <bool expression> | <bool expression> <guard op>
256     <guard>

257  <guard op> ::= cand | cor

258 *<bool expression> ::= <bool factor> |
259     <bool factor> <combining op> <bool expression>

260 *<bool factor> ::= <bool primary> | <negating op> <bool primary>

261 *<bool primary> ::= <object ref> | <constant> |
262     <program activation> | ( <bool expression> ) |
263     <arith expression> <relational op> <arith expression>

264 *<constant> ::= <character literal> | <boolean literal> |
265     <numeric literal> | <array literal>

266  <character literal> ::= " <character list> "

267  <numeric literal> ::= <integer> | <integer> . <number> |
268     <integer> E <integer> | <integer> . <number> E <integer>

269  <array literal> ::= ( <number> , <array value list> )

270  <character list> ::= <ascii char> | <ascii char> <character list>

271  <ascii char> ::= {any printable character except "{" and "}"}

272  <array value list> ::= <constant> | <constant> , <array value list>

273 *<integer> ::= <sign> <number> | <number>

274  <sign> ::= + | -
```

```
275 *<number> ::= <digit> | <digit> <number>
276  <digit> ::= {"0" - "9"}
277 *<identifier> ::= <letter> | <letter> <alphan list>
278  <alphan list> ::= <alphan> | <alphan> <alphan list>
279  <alphan> ::= <letter> | <digit>
280  <letter> ::= {"A" - "Z", "a" - "z", "_"}
281  <nil> ::= {nothing}
```

B.8 Nonterminal Cross Reference

<CSDL text> 1:

<action decl> 95: 94

<action def> 97: 95, 96

<action id> 108: 95, 96, 217

<action list> 94: 93, 94

<action ref> 217: 212, 213, 215

<actions section> 93: 19

<actual parameter list> 184: 181, 185, 217, 228

<actual parameter> 231: 185

<adding op> 243: 241

<afactor> 169: 167, 168, 169

<alphan list> 278: 277, 278

<alphan> 279: 278

<alternative construct> 224: 222

<arith expression> 241: 91, 143, 205, 210, 237, 238, 239, 241, 250, 251, 262, 263

<arith factor> 247: 244

<arith primary> 249: 247

<arith term> 244: 241, 244

<array literal> 269: 265

<array type> 64: 58

<array value list> 272: 269, 272

<ascii char> 271: 270

<assertion> 160: 36, 37, 39, 40, 70, 71, 77, 122, 123, 145, 161

<assignment statement> 223: 220